



CLOUD-BASED FPGA CUSTOM COMPUTING MACHINES

BY

AMRAN ABDULRAHMAN AL-AGHBARI

A Dissertation Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

DECEMBER 2018

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Amran Abdulrahman Al-Aghbari** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING.**

 13/2/2019


Dr. Mohamed Elrabaa
(Advisor)



Dr. Adel Fadhl Ahmed
Department Chairman

 13/2/2019

Prof. Aiman El-Maleh
(Member)

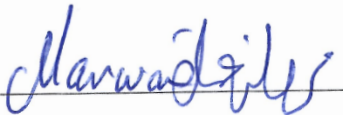

Dr. Salam A. Zummo
Dean of Graduate Studies



 21/11/2018

Dr. Mohammad Alshayeb
(Member)

19/2/19
Date



Dr. Marwan Abu-Amara
(Member)



Dr. Ashraf S. Hasan Mahmoud
(Member)

© Amran Abdulrahman Al-Aghbari

2018

Dedication

To my wife, children, and our parents, brothers, sisters and their children.

ACKNOWLEDGMENTS

Full thanks to my advisor Dr. Mohamed Elrabaa who guided me along my graduate studies at KFUPM University and taught me VLSI from scratch until fabricating a real chip. Thanks to Prof. Mayez Al-Mohamed who taught me most of the valuable principles I know about computing architectures and parallel computing which were helpful in this work. Thanks to all professors who taught me different courses during my graduate studies which were valuable guidance to accomplish this work. Thanks to KFUPM University which provides those expensive devices and licenses that helps a lot to do real and valuable experiments. Thanks to Saudi Arabia for financial support for their students and foreign students. Thanks to Taiz University, my home university in my home country for sending me to KFUPM University to do graduate studies and supporting me financially.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VI
LIST OF TABLES	X
LIST OF FIGURES	XII
LIST OF ABBREVIATIONS	XVII
DISSERTATION ABSTRACT	XIX
الدكتوراة في الفلسفة ملخص بحث درجة	XXI
CHAPTER 1 INTRODUCTION	1
1.1 Motivation	1
1.2 Thesis statement and contributions	5
1.3 Overview of the thesis	6
CHAPTER 2 CLOUD COMPUTING	8
2.1 Data center	10
2.2 Virtualization	12
2.2.1 Hardware virtualization	12
2.2.2 Application virtualization	13
2.2.3 Desktop virtualization	13
2.2.4 Network virtualization	14
2.2.5 Storage virtualization	14
2.3 OpenStack: open source cloud computing platform	15
CHAPTER 3 DESIGN WITH FPGAS	17
3.1 FPGA architecture	18
3.2 Design flow	20

3.3	Partial Reconfiguration	21
3.4	High-level synthesis	23
3.5	FPGA strengths and weaknesses	25
CHAPTER 4 LITERATURE REVIEW		28
4.1	Overlay architecture	28
4.2	Virtualization using abstraction layer (AL).....	30
4.2.1	Interfacing the abstraction layer (AL) with vFPGAs.....	32
4.2.2	OpenCL and the CPU-FPGAs interface abstraction	33
4.3	FPGA attachment interface	35
4.4	FPGA in the cloud and data center	37
4.5	ASIC Clouds	39
4.6	Summary.....	40
CHAPTER 5 OVERVIEW OF THE CLOUD-BASED FPGA CUSTOM COMPUTING MACHINES PLATFORM		43
5.1	FPGA Virtualization	44
5.2	FPGA Cloud Architecture	47
5.3	FPGA hypervisor	48
5.3.1	User-to-CCM API functions.....	50
5.3.2	User-to-Hypervisor API functions.....	51
5.3.3	Hypervisor-to-Hypervisor back-end API functions	52
5.4	A scenario of Launching, Using and Terminating a CCM	53
5.5	CCM Creation	55
5.6	Properties of the Platform	56
5.6.1	The platform computing model	56
5.6.2	Abstraction	57
5.6.3	Sharing.....	58
5.6.4	User data security	58
5.6.5	CCM clusters on Multi-vFPGA	58
CHAPTER 6 FPGA VIRTUALIZATION PLATFORM		60

6.1	Data Communications	61
6.2	Network Controller	64
6.3	Static logic	66
6.3.1	Data routing	66
6.3.2	Reconfiguration management unit (RM)	67
6.3.3	Clock management unit (CM).....	67
6.4	The wrapper's design	68
6.4.1	Conceptual design of the wrapper	70
6.4.2	Wrapper components	71
6.5	Wrapper generation	77
6.5.1	Parsing the XML/JSON specification file.....	79
6.5.2	Parsing the Vera specification file	84
6.5.3	An example for generating a serializer from a Vera description.....	85
6.5.4	Wrapper generation software.....	88
 CHAPTER 7 RESULTS AND COMPARISON		89
7.1	Generating a wrapper for the JPEG Encode core	90
7.1.1	Preparing the XML Description File.....	90
7.1.2	User's Vera Data Specifications.....	91
7.1.3	JPEG Encoder implementation on a vFPGA.....	93
7.2	Simulation methodology	96
7.3	Virtualization Overhead Evaluation	98
7.4	Comparisons with other platforms	104
7.5	vFPGAs versus SW-based virtual machines	106
7.6	CCM platform Evaluation	109
7.6.1	Experiment setup	109
7.6.2	Performance Evaluation	113
7.6.3	The impact of adding the AES encryption/decryption	115
7.6.4	The impact of having multiple vFPGAs within the same FPGA	117
7.7	Boot time analysis	118
 CHAPTER 8 CONCLUSION		120
8.1	Platform Limitations	121
8.2	Future work	122

APPENDICES	125
A. Description of the used Benchmarks	125
A.1. JPEG Encoder Core	125
A.2. AES Core	126
A.3. RSA512 Core	128
A.4. FDCT Core	130
A.5. JPEG Images Edge Detection	132
A.6. Decrypt-Compute-Encrypt	133
B. Software tool for Ethernet packet generation and platform test bench	134
C. A description of the implemented Verilog code	137
REFERENCES	139
VITAE	148

LIST OF TABLES

Table 4-1:	List of notable platforms of FPGA-based processing for clouds or datacenters.	42
Table 5-1:	Main API function in the software library.....	50
Table 6-1:	A description of the XML tags and their attributes used to describe the user hardware I/Os and their groups.....	81
Table 6.2:	FDCT benchmark verification code written on OpenVera and its translation to microcode microinstructions.....	87
Table 7.1:	Formatting and applying the JPEG Encoder’s input data by the wrapper. Four control bits are added with each input. The last column represents the complete output of the serializer which is applied to the input FIFO.....	93
Table 7.2:	Virtualization overhead compared to direct implementation on an FPGA for 4 benchmarks. For the vFPGAs, the wrapper’s I/O widths are 64/64 bits for all designs.....	100
Table 7.3:	Comparison with notable platforms for attaching FPGAs to DCs.	105
Table 7.4:	Resource utilization of the virtualization platform on FPGA.....	112
Table 7.5:	Computation time comparison for three implementations of the secure image edge detection application. a) The application on a virtual machine, b) The application on a server, c) The application is a CCM.....	115
Table 7.6:	Estimation of the CCM area overhead in our platform.	117
Table 7.7:	Boot time delay components for vFPGA-CCMs with various image (bitstream) sizes. Internal configuration access port’s speed is ~400MB/s	119
Table 8.1:	Snapshot of the serializer’s output for the JPEG encoder. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register and the clock counter. Then it sets values to the least frequently used signals. After that, the data starts.	126
Table 8.2:	Snapshot of the serializer’s output for the AES128. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register and the clock counter. Then it sets the key. After that, the data starts.	128
Table 8.3:	Snapshot of the serializer’s output for the RSA512. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register. It then set the value of the modulus m . Then, it sets the <i>bit_size</i> constants. After	

that, it sets the clock counter to 32 cycles. Then, it applies those clocks to the design. After that, it resets the clock counter to zero. Then, the data starts.....130

Table 8.4: Snapshot of the serializer's output for the FDCT. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register. After that, it resets the clock counter to zero. Then, it sets the enable and reset signals high and the dstrb signal low. After that, the first byte of the data is set. Then, it sets the dstrb signal high. After that, it sets the dstrb signal low again. Then, the data continues.....132

LIST OF FIGURES

Figure 2.1:	Data center architecture has three layers.	11
Figure 2.2:	OpenStack core components [13].	15
Figure 3.1:	FPGA Architecture is a two-dimensional array of reconfigurable resources. Components on FPGA are programmable: control logic blocks (CLBs), SRAMs, DSP blocks, and interconnects [14].	18
Figure 3.2:	3-input LUT implementations	19
Figure 3.3:	Compute paradigms comparison [28]. a) CPU represents the Von Neumann model. b) GPU represents the vector processing model. c) FPGA represents spatial computing.	26
Figure 3.4:	GPU vs FPGA qualitative comparison [5].	27
Figure 4.1:	The architecture and design components in Intel OpenCL for FPGAs [14].	35
Figure 5.1:	FPGA virtualization is based on several abstraction layers.	44
Figure 5.2:	Proposed FPGA Cloud Architecture	47
Figure 5.3:	Python implementations for the functions “Send (data stream)” and “Listen_to_results (data stream)”. Both functions use TCP stream socket and require the CCM IP address and port number.	51
Figure 5.4:	Hypervisor to hypervisor’s backend functions uses UDP socket connection. The UDP payload contains a sequence of CMD and value pairs. Several commands can be sent on one UDP packet. Hypervisor sends a UDP packet and hypervisor’s back end reply with a UDP packet.	53
Figure 5.5:	Scenario of using a CCM on a cloud computing system. The user issues four commands to launch, send data, receives results, and terminate CCM.	53
Figure 5.6:	Scenario of using a CCM on a cloud computing system. First, the user requests to launch a CCM. The FPGA hypervisor configures a vFPGA with the CCM bitstream and returns the CCM IP address back to the user. The user interacts with the CCM by sending data and receiving results. Finally, the user releases the CCM.	55
Figure 5.7:	CCM creator receives hardware design (HLS/HDL), XML file describes hardware I/Os, and Vera file describes how data is applied. Then, it creates a CCM, synthesized and generates partial bitstreams. Finally, it saves the bitstreams in the cloud storage and their info in the Resources database.	56
Figure 6.1:	Virtualization platform overview. FPGA is partitioned into a static region and several reconfigurable regions to be used as virtual FPGAs.	61
Figure 6.2:	Timing diagram of the two-way handshaking process [48].	62

Figure 6.3:	BRAM-based asynchronous FIFO for transferring data across unrelated clock domains using the AXI interface.....	62
Figure 6.4:	Inter-layer interfaces. The interface between L1 and L2 consists of two AXI interfaces and the virtual FPGA indices. The interface between L2 and L3 consists of two AXI interfaces and clocking signals. The interface between L3 and L4 consists of two AXI interfaces whose data have the internal wrapper formats.....	63
Figure 6.5:	The implementation of the Ethernet controller.....	66
Figure 6.6:	The platform's different clock domains and the use of asynchronous buffers to move data across these domains.....	68
Figure 6.7:	A flow chart illustrating the data flow from/to the design through the wrapper. The left-hand side shows the data input flow starting from receiving a payload of a user's network packet till its application to the design. The right-hand side shows formatting and sending the results starting from capturing the outputs till generating the payload for the network packet to be sent back to the user.	71
Figure 6.8:	Our implementation of the Packing/unpacking circuitry. If the input data width is greater than the output data width, the packing circuitry is used. If it is less, the unpacking circuitry is used. If they are equal, the packing/unpacking part is removed.	72
Figure 6.9:	Slicer is a combinational circuit that selects $n/2$ consecutive bits from an n -bit input.....	73
Figure 6.10:	The wrapper's conceptual design.	74
Figure 6.11:	Diagram shows the complexity of building the wrapper state machine. If the hardware has one output group, then an input can be applied while capturing the output s . If the hardware has several outputs, then the controller should flush out outputs before accepting new inputs. For some inputs, it is required to apply several consecutive clock cycles without capturing outputs or applying new inputs.....	75
Figure 6.12:	Verilog code of a finite state machine of a wrapper.	76
Figure 6.13:	The controllable clock buffer allows controlling the application clock. When it is enabled the application run. When it is disabled the application freezes. The upper timing diagram shows a clock buffer which always produces a low output when its enable signal is off. The lower timing diagram shows a clock buffer which always produces a high output when its enable signal is off.	77
Figure 6.14:	CCM creation flowchart.	78
Figure 6.15:	Algorithm for generating the wrapper from the XML specification file and the Vera description file.....	79
Figure 6.16:	Algorithm for parsing the XML specification file and generating Verilog code for the modifiable parts of the wrapper.	80

Figure 6.17: The JSON schema file for describing hardware I/Os and their groups to the wrapper generator.	82
Figure 6.18: The XML schema file for describing hardware I/Os and their groups to the wrapper generator.	83
Figure 6.19: The OpenVera code is translated to microinstructions then the Serialized is generated.	84
Figure 6.20: The FDCT I/O specification in the XML file.	85
Figure 6.21: The serializer for the FDCT can be generated automatically from the Vera specification code Using a microcode-template. Each cycle of the microcode generates data for one input group and generates the group index and one bit represents whether to apply a clock or not for this data.	87
Figure 6.22: A snapshot of the wrapper builder software. The list on the left contains several hardware cores. The wrapper is generated instantly for the selected hardware core. The bottom large textbox contains the generated wrapper Verilog code.....	88
Figure 7.1: Generating the wrapper for the JPEG Encoder from the XML and Vera specifications.	90
Figure 7.2: Snapshot of the complete wrapper's and the Encoder's input/output and control signals.	95
Figure 7.3: Packing the 8-bit wrapper inputs into the Encoder's 28-bits inputs in ~28/8 cycles per input. e.g. input sequence 7F FF FF D4 00 00 00 is packed into 7FFFFFFD 4000000 sequence.	96
Figure 7.4: Unpacking the Encoder's 39-bits outputs to produce 8-bit wrapper's output per cycle.....	96
Figure 7.5: Simulation methodology to simulate the whole platform. The simulator inputs are Ethernet packets. The simulator outputs are Ethernet packets.....	97
Figure 7.6: The simulation of the FDCT core as it is designed by the core designer. The total computation time is measured to be 175,811.2 nanoseconds.....	97
Figure 7.7: The simulation of the FDCT core placed within a vFPGA in the implemented virtualization platform (using 10GE). The time from receiving the first Ethernet packet (RXDV changes) until the last Ethernet packet is transmitted out (TXEN changes) is measured to be 198,860.6 nanoseconds.	98
Figure 7.8: Wrapper area versus the number of the applicaion I/Os for 1, 2, 3 grouping.	102
Figure 7.9: The XML specification of the input/output groups of a black box (one group, two groups, and three groups). The black box has no design inside. It is used to generate a wrapper for an assumed design with arbitrary inputs/outputs and an arbitrary number of groups. The black box is used	

to evaluate the wrapper area for different number of inputs/outputs and different number of groups.	103
Figure 7.10: The three platforms used to evaluate the performance of a streaming application; (a) Running in a virtual machine, (b) directly on the physical server, and (c) on a vFPGA. A client SW sends encrypted data and receives encrypted computation results.	107
Figure 7.11: Streaming application throughput versus block size comparisons the proposed vFPGA platform and physical servers and virtual machines.	109
Figure 7.12: The experimental setup with several versions of the secure edge detection (ED) application.....	110
Figure 7.13: Synchronization process among the sender, receiver and the server. The server manages to start and to end the work in the three steps at the same time.	111
Figure 7.14: The FPGA virtualization platform with the Edge detector application implemented as a CCM.....	111
Figure 7.15: Image edge detection hardware uses four already-made cores; AES 128 [72], Image Compress [69], Canny Edge Detection [78], and JPEG encoder [70].	112
Figure 7.16: The software version of the application “Secure image edge detection” written in Python using standard SW libraries	113
Figure 7.17: The user uses the same socket interface to request the same service hosted n three different machines; a) the service is hosted in a VM, b) the service is hosted in a server, c) the service is a CCM on virtual FPGA.	114
Figure 7.18: Compute nodes performance comparison for a specific application. vFPGA outperforms a virtual machine and a bare-metal server.	115
Figure 7.19: Using the AES-ECB core [72] to build AES-CTR that can be used as a decrypter and encrypter. By XORing the input text with the encrypted counter output we achieve a throughput of one data block per cycle. AES-CTR throughput is one block per cycle because the XORing takes one cycle only.....	116
Figure 7.20: Different boot time components of a vFPGA-CCM.....	118
Figure 8.1: The XML specification of the input/output groups of the jpeg encoder core [70].....	126
Figure 8.2: The XML specification of the input/output groups of the jpeg encoder core [72].....	127
Figure 8.3: The XML specification of the input/output groups of the rsa512 core [71].....	129
Figure 8.4: The XML specification of the input/output groups of the DCT core [69].	131

Figure 8.5: The XML specification of the input/output groups of the image edge detection we designed by combining several cores.	133
Figure 8.6: The XML specification of the input/output groups of the decrypt-compute-decrypt hardware designed using the AES-CTR which uses the AES-ECB core [72].	133
Figure 8.7: Generate platform test bench.	134
Figure 8.8: Make UDP header algorithm. The algorithm is inspired by the IP formal definition in RFC 791 [83] and the UDP formal definition in RFC 768 [84].	135
Figure 8.9: Calculate IP checksum algorithm. The algorithm is inspired by the IP checksum calculation description.	136
Figure 8.10: Calculate UDP checksum algorithm. The algorithm is inspired by the UDP checksum calculation description in the UDP formal definition in RFC 768 [84].	136
Figure 8.11: A snapshot shows the hierarchy and components of an implemented version of the virtualization platform. The hierarchy starts the root node “top_xge” which contains clocking resources appears in the first four nodes and the virtualization module “virt005” which contains the platform and the network controller. The platform contains data routers (mux and demux), the reconfiguration module (uses ICAP) and one vFPGA. The vFPGA contains the image_edge_detect application which uses four already-made cores; AES_128 [72], jpeg_decode [77], top_edge [78] and jpeg encoder [70].	138

LIST OF ABBREVIATIONS

AL	:	Abstraction Layer
ARP	:	Address Resolution Protocol
ASIC	:	Application Specific Integrated Circuit
AXI	:	Advanced eXtensible Interface
BRAM	:	Block RAM
CCM	:	Custom Computing Machines
CM	:	Configuration Manager
CPU	:	Central Processing Unit
DC	:	Data Center
DHCP	:	Dynamic Host Configuration Protocol
DPR	:	Dynamic Partial Reconfiguration
DSP	:	Digital Signal Processor
FF	:	Flip-Flop
FPGA	:	Field-Programmable Gate Array
GPU	:	Graphic Processing Unit
GMI	:	Gigabit media-independent interface
HLS	:	High Level Synthesis

HW	:	Hardware
IOB	:	Input Output Buffer
IP	:	Internet Protocol
LUT	:	Lookup Table
MII	:	Media-independent interface
PCIe	:	Peripheral Component Interconnect Express
PR	:	Partial Reconfiguration
SDN	:	Software-defined networks
SW	:	Software
TCP	:	Transmission Control Protocol
UDP	:	User Datagram Protocol
vFPGA	:	Virtual FPGA
VM	:	Virtual Machine
XGMII	:	10 gigabit media-independent interface

DISSERTATION ABSTRACT

Full Name : Amran Abdulrahman Al-Aghbari.
Thesis Title : Cloud-based FPGA custom computing machines.
Major Field : Computer science and engineering.
Date of Degree : December 2018

Field Programmable Gate Arrays (FPGAs) were first introduced as large capacity platforms of glue logic used for logic emulation and prototyping. Later, research efforts explored the use of FPGAs as computing devices. FPGAs provide excellent performance for several application domains, achieve lower power per operation, provide more deterministic latency, and can be connected to hosts or other FPGAs using different types of interfaces. On the negative side, FPGAs have a long design time in comparison with other computing machines such as CPUs and GPUs. Designing a hardware application, debugging it, and verifying its correctness requires hours and days. The resulting hardware application is both a vendor and a device-dependent. As such, developing an FPGA-based HW application is still restricted to hardware designers. For others to develop HW applications on FPGAs, the FPGA has to be properly virtualized, and its interfaces abstracted. Virtualization also enables integrating FPGAs within computing infrastructures such as data centers and clouds. In this dissertation, we introduce an FPGA virtualization platform that enables any application hardware to be ported on virtual FPGA and accessed as a standalone custom computing machine (CCM). We propose an FPGA cloud computing platform that introduces CCM as a service. A prototype of the virtualization platform has been implemented to evaluate its area, speed, and overhead. Comparison with

other platforms shows that the proposed platform provides a general abstract interface to any design (not domain specific) and supports dynamic partial reconfiguration (so designs can be added to an FPGA that have other applications running) at comparable overhead to other notable platforms. Experimental results, using a streamed application in a cloud-like environment, showed that the proposed platform is a very viable computing option (in terms of throughput) for some applications compared to conventional server-based or virtual-machine based SW implementations.

ملخصُ بحث درجةُ الدكتوراةِ في الفلسفةِ

الاسم الكامل : عمران عبد الرحمن عبد الولي الأغبري

عنوان الرسالة : آلات حوسبة متخصصة باستخدام الرقاقات القابلة للبرمجة (FPGAs) و دمجها في الحوسبة السحابية

التخصص : علوم و هندسة الحاسب الآلي

تاريخ الدرجة العلمية : ديسمبر 2018

عرفت المصفوفات القابلة للبرمجة في بداياتها بأنها منصات ذات كمية كبيرة من الدوائر الرقمية و استخدمت في محاكاة الأجهزة الرقمية و عمل نماذج لها. الجهود البحثية الأخيرة بحثت في استخدام المصفوفات القابلة للبرمجة كأجهزة حوسبة و إمكانية استخدامها لتنفيذ برمجيات. لقد أظهرت المصفوفات القابلة للبرمجة، أداءً عالياً في العديد من المجالات، كما أظهرت أنها أقل استهلاكاً للطاقة لكل عملية حسابية، و الزمن اللازم للحساب فيها أكثر قابليةً للتحديد من غيرها، بالإضافة لقابلية ربطها بالأجهزة الأخرى عبر أنواع عديدة من المنافذ. من جهة أخرى، الوقت المستهلك في تصميم البرامج للمصفوفات القابلة للبرمجة طويل مقارنةً بأجهزة الحوسبة الأخرى كمعالجات الحواسيب أو معالجات الرسوميات. إن تصميم برنامج على المصفوفات القابلة للبرمجة و التأكد من صحة عمله قد يستهلك ساعات و أيام. و هذا البرنامج لا يعمل إلا على نوعية المصفوفات القابلة للبرمجة التي تم تصميمه عليها. و بالتالي، تصميم البرامج على المصفوفات القابلة للبرمجة لا يزال حكراً على المختصين فيها فقط حتى الآن. بالنسبة لغير المختصين، من الضروري تصميم مصفوفات قابلة للبرمجة افتراضية كما ينبغي تبسيط طرق الاتصال بها. تصميم مصفوفات قابلة للبرمجة افتراضية سوف يعطي أيضاً إمكانية وضعها ضمن أجهزة الحوسبة الأخرى في الحوسبة السحابية و مراكز البيانات. في هذه الأطروحة، سوف نقدم مقترح منصة تحتوي على مصفوفات قابلة للبرمجة افتراضية، و التي سوف تتيح لمصممي البرامج الرقمية وضع برمجياتهم في مصفوفة افتراضية و التواصل معها كجهاز حوسبة مستقل. سنقدم أيضاً مقترح منصة للمصفوفات القابلة للبرمجة على الحوسبة السحابية، و الذي سوف يتيح تقديم خدمة سحابية

نسميها خدمة جهاز حوسبة متخصص. قمنا بتنفيذ نموذج للمصفوفات القابلة للبرمجة الافتراضية بغرض تقييمها من حيث المساحة و السرعة و الكلفة، و قمنا بمقارنته مع المقترحات البحثية الشبيهة. المقارنة أظهرت أن المنصة التي نقتريها تعطي طرق اتصال أعم و بالتالي يمكنها قبول أية برامج و هي غير مقتصرة على مجال محدد من البرمجيات و كلفتها مقبولة مقارنة بالأعمال المشابهة. كما أن المنصة تدعم التركيب الجزئي في المصفوفات القابلة للبرمجة و هو يعني أن المصفوفة الواحدة يمكن تقسيمها إلى عدة منصات افتراضية يمكن أن يتم تحميل البرامج على أي منها دون الحاجة لتوقيف جميع المنصات. في التجارب العملية، استخدمنا برامج تستخدم بيانات مستمرة في بيئة تشبه بيئة الحوسبة السحابية. أظهرت النتائج أن المنصة المقترحة هي خيار ذو قيمة عالية جدا في مجال الحوسبة من ناحية الانتاجية لبعض التطبيقات مقارنة بالخوادم و الآلات الافتراضية المستخدمة حاليا كمنصات لتنفيذ البرمجيات في الحوسبة السحابية و مراكز البيانات.

CHAPTER 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) were first introduced as large capacity platforms of glue logic used for logic emulation and prototyping. Later, research efforts explored the use of FPGAs as computing devices. FPGAs reports excellent performance for AI, image processing, data compression, and other applications. By using FPGA accelerator attached to CPUs, different applications in pattern matching can become 300+ times faster, 200+ times faster for compression, 100+ times faster in machine learning and more [1]. Intel announced three powerful FPGA-based accelerator libraries [2]; 1) Intel's Convolutional Neural Networks (CNN) engine for FPGAs. 2) Real-time data analytics algorithms. 3) Data compression algorithms with dynamic compression ratios. Different Azure network relies on FPGA-powered software-defined networking (SDN) [3]. The bandwidth between two VMs inside Azure, with a 40-gigabit network adapter on each VM,

is only around 4Gbps per second; with FPGA-accelerated networking, that goes up to 25Gbps, with five to ten times less latency [3].

Excellent performance for several applications is not the only motivation for using FPGAs in computation. FPGAs are the least power consumers per operation among existing processing units such as CPUs and GPUs [4, 5]. Real-time applications prefer FPGA because their latency is deterministic with accuracy reaches to nanoseconds. FPGA can be programmed to deal with any type of interfaces and can be offered as a standalone computing device.

Unfortunately, FPGA programmability is the worst among other computing devices such as CPUs and GPUs [5]. Compiling an application code written in hardware description languages (HDL), the standard method to write designs for FPGAs, to FPGA configuration bitstream takes minutes and maybe hours (a typical FPGA design flow is explained in chapter 3). Hardware debugging and verification usually require several design changes, synthesis (compilation), and simulation that cost hours and maybe days. Due to this long design and compilation time, hardware designers tend to produce their designs as hardware cores which are simulated, verified, tested and then introduced as black boxes. Those hardware cores could be used directly for computation or as a building block within other hardware cores.

Although using FPGAs for computation succeeded for several application domains, their usage is still limited to hardware designers. Working with FPGAs requires a hardware-background which prevents many users from using FPGAs for computation purposes. A lot of work is required before having FPGAs available on the cloud for the mass. First,

FPGAs must be virtualized to be a cloud resource. Virtual FPGAs reveals the user from implementation details. There are a lot of physical details that need to be hidden from the FPGA user. FPGAs vary in their architecture, capacity, vendors, clocking resources and frequency. There are common tasks that are needed for all applications such as FPGA programming, clocking management, securing data and interfacing FPGA with the ecosystem. FPGA virtualization hides physical details and automates the frequently needed tasks. Second, the FPGA interface must be abstracted to enable FPGA to interact with the ecosystem smoothly. Each hardware core has its own interface and interfacing protocol. Using the hardware core requires following the core specifications provided by its designer. FPGA interface needs to be abstracted such that it provides FPGA that can easily hold any hardware core and let it work smoothly. Third, standard software libraries should be provided as a mechanism to interact with applications in FPGAs. FPGA should be able to work easily with the data structures used in the ecosystem.

There is an increasing trend for using FPGAs in cloud and data centers since they provide better utilization with low power compared with the current CPU-based servers which consume excessive power with low utilization. Microsoft uses FPGAs to accelerate the Bing search [6]. It uses FPGA-based SDN to accelerate its networking operations [2]. Amazon introduced FPGA infrastructure as a service [7] two years ago. The increasing trend for using FPGAs reveals the need to virtualize FPGAs and introduce an easier method to interact with them. FPGA virtualization is the first step to introduce them as cloud computing resources. Two common approaches have emerged for attaching FPGAs to data centers (DCs); as accelerators attached to compute nodes via a local bus such as the Peripheral Component Interconnect Express (PCIe), or as stand-alone independent

computing resources connected to the DC's interconnect fabric (i.e. Ethernet LAN). Since cloud computing resources are network-attached nodes, FPGAs should be network-attached devices rather than PCIe-attached ones. FPGAs should be disaggregated from CPUs, dealt with as standalone computing machines and provided as standalone cloud computing resources.

To summarize, the main motivation behind this work is to facilitate the use of custom application hardware in typical computing infrastructures such as data centers and clouds. This requires the following:

- 1) A method for deploying HW applications on any network-attached FPGA without the need to re-design or re-synthesize the application for different FPGAs,
- 2) A method for completely abstracting hardware interfaces to enable accessing them as standalone computing machines (similar to SW application servers). Abstraction layers should be clearly identified and auto-generation tools for these layers should be provided,
- 3) The overheads (cost/area, performance, and power) resulting from these abstraction layers must be evaluated to determine the feasibility of the whole approach,
- 4) Finally, even with virtualization and abstraction, current FPGAs are not general computing platforms. Hence suitable applications and execution models for the developed techniques must be identified.

1.2 Thesis statement and contributions

The notion of building a custom computing machine (CCM) appeared to indicate creating special hardware for a specific computing task on FPGA. There are a lot of already-designed hardware cores that efficiently do computations on FPGA such as crypto cores, image and video processors, arithmetic cores and machine learning cores. Some of these cores are open-source cores provided in websites such as “github.com” and “opencores.org” while other cores require licenses from the provider. Each core could be introduced as a standalone CCM accessible over the network by users with a non-hardware background. Whatever the design flow used to build hardware cores, there should be a flexible platform capable of hosting them, abstract their interfaces and deploy their services to the mass. This platform should provide flexibility by hiding hardware complexities and restricting the input method to well-known data formats used in the ecosystem such as text, images, and video streams. It should provide low power consumption since it uses FPGAs only without the help of an external CPU-based controller. The hardware core should also maintain its high performance when it is hosted on the platform. The platform should virtualize FPGAs, abstract their interface and enable integrating them on data centers and introducing them as cloud services. The process of accessing and using CCMs should be automated. The resulting CCM should interact with the user using a software library that can be integrated with the high-level programming language (HLL) used by the user.

The goal of this dissertation is to develop a new methodology for using FPGA for computation in cloud and data centers. Here, we outline the contributions of this dissertation as follows:

- We propose the FPGA custom computing machine (CCM) as an abstraction of application hardware for cloud and data centers. CCM hides hardware complexities and restricts the application hardware input and output to well-known data formats used in the ecosystem.
- We introduce a cloud platform for virtual FPGA resources management and introducing CCM-as-a-service. We introduce the software library for launching, using, releasing CCMs.
- We propose an FPGA virtualization platform which provides network-attached virtual FPGAs. We clearly illustrate the physical FPGA interface and the virtual FPGA interface.
- For the virtual FPGA interface, we introduce our wrapper design. The wrapper is a circuit that should be added to the user hardware to adapt its interface to match the vFPGA interface. It abstracts the data movement with the application hardware. It can be added to any already-made hardware core and make it a CCM without any internal modifications. The wrapper is auto-generated from the hardware core specifications. We also illustrate the auto-generation process.

1.3 Overview of the thesis

The thesis is organized in the following manner. Chapter 2 gives background about cloud computing principles and the main virtualization techniques used in the cloud; virtual machines and containers. The chapter discusses OpenStack, the open source cloud computing platform, which is used for integrating FPGAs in the cloud. Chapter 3 gives background on HW application design using FPGAs. It explores FPGA architectures, its

components, and the hardware description languages (HDL) as the main method to write hardware designs. It also discusses high-level synthesis (HLS) tools and describes the most notable ones. Chapter 4 reviews the state of the art in FPGA virtualization, interface abstraction, integration with data centers and cloud management frameworks. A special review of OpenCL is also provided as it completely abstracts using FPGAs for computation. Chapter 5 presents an overview of the proposed cloud platform that introduces CCM-as-a-service. The FPGA hypervisor with the software library is discussed and illustrated with use-case scenarios. CCM creation process is also discussed and also chaining several CCMs to use them as one CCM. Chapter 6 discusses the FPGA virtualization platform with physical implementation details. Chapter 7 presents results and comparison. It starts with a test case that explains creating the wrapper and how it works. Then, the virtualization platform is evaluated and the overhead is reported in terms of area, performance, throughput, and power with several hardware cores. The CCM platform is evaluated with an edge detector hardware application. A comparison with other works is stated. Finally, chapter 8 presents a conclusion.

CHAPTER 2

Cloud Computing

Cloud computing is a model for enabling network access to a shared pool of resources. These resources can be provisioned and released with minimal service provider interaction. As stated by NIST [8], cloud computing reveals the following fundamental characteristics:

- *On-demand self-service*: New resources are provisioned without human interaction.
- *Network access*: Resources are accessed over the network by standard mechanisms.
- *Resource pooling*: Resources are abstracted into pools to serve multiple users dynamically.
- *Rapid elasticity*: The amount of resources is expanded and shrunk smoothly so that it appears to the customer unlimited.
- *Measured service*: Resource usage is metered, controlled and reported.

Cloud computing provides services using one of these models:

- *Software as a Service (SaaS)*: Provides application software, databases, support.
Example: Google Applications.

- *Platform as a Service (PaaS)*: Provides Operating system, development, and execution framework, database, web server. Example: Microsoft Azure.
- *Infrastructure as a Service (IaaS)*: Provides virtual machines, storage, and connectivity. Example: Amazon Elastic Compute Cloud (Amazon EC2).

Cloud computing provides us with several benefits including the following:

- Cloud computing is a cost-efficient system and it has a pay-per-use billing model. The maintenance is lower than that in traditional computing since the infrastructure is not purchased.
- Cloud computing systems offer large storage and handle their maintenance, backup, and recovery.
- Cloud computing offers a more flexible solution and adapting applications rapidly with the changes on business conditions.

On the other side, cloud computing faces several challenges that must be taken into consideration when dealing with them:

- Data security and privacy are big concerns on cloud computing. Service providers provide data security and privacy the user should rely on them.
- Cloud services management is not trivial. Data replication and recovery, capacity management, auto-scaling, and transactions monitoring all have to be served sufficiently to avoid damage and severe impacts.
- Cloud computing systems must obey some government regulations in many cases. For example, some governments do not allow having storage systems outside the country that might store people personal information.

2.1 Data center

Data center (DC) is a collection of servers interconnected by switches within a room/building. DC provides large storage, large-scale services that process a large amount of data and serve a large number of users. Large companies such as Facebook, Microsoft, and Google have their own DCs. Small companies rent DC resources through cloud computing services. Within the data center building, we find the following components:

- IT devices: The IT devices can be categorized into servers, communication devices, and storage.

A server is a device that provides functionalities, called "services", for other devices called "clients". In the client-server model, computations are distributed across servers. Typical servers are database servers, file servers, mail servers, print servers, web servers, game servers, and application servers. The servers of the data center are stacked in racks that are placed in rows and called server farms.

The storage in data centers is usually disaggregated from the servers as a storage or disk array. A disk array is a disk storage system consists of multiple disk drives and cache memory and supports virtualization and Redundant Array of Independent Disks (RAID).

- Mechanical and electrical infrastructures: data centers require a cooling system for the IT devices. It also requires redundant electricity. The electrical infrastructure provides power without interruption for the IT devices. A backup generator provides reliable power in case of power outage. An uninterruptible power supply (UPS) ensures that the quality remains constant even after a power outage.

The data center components are well organized in racks and cabinet. A rack is a physical steel that is designed to house servers, networking devices, and other computing equipment. It is prefabricated with slots for connecting cables. Data center racks are classified based on their dimensions and capacity, the amount of equipment they can hold.

The data center architecture has three layers; core, aggregation and access (edge), Figure 2.1. The core networks contain the border routers that connect the data center to the internet. The aggregation layer provides functionalities for routing, load balancing, firewalls, intrusion detection, traffic flow, and more. The access network contains storage, servers, and the inexpensive switches that access the servers.

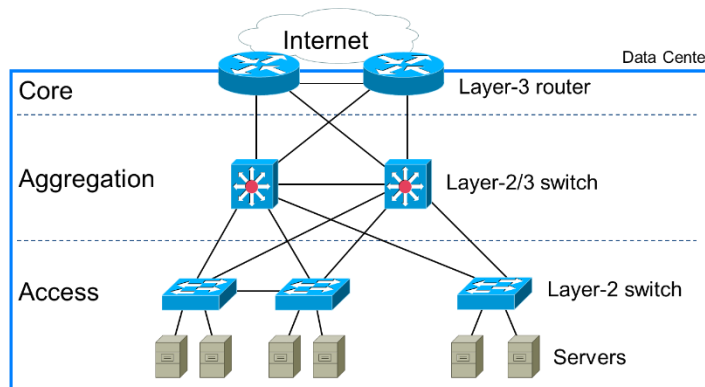


Figure 2.1: Data center architecture has three layers.

There are two design models for the data centers; the multi-tier model and the server cluster model. The multi-tier model divides the data center servers into three tiers of servers: web-servers tier, application servers tier, and database servers tier. This model is used to provide web services and HTTP-based applications. The server cluster model combines several CPUs as a unified high-performance system using a high-speed network. This model is used for high-performance computing (HPC), parallel computing, and grid computing.

2.2 Virtualization

Virtualization is the creation of a virtual version of something. All IT equipment in data centers and cloud are virtualized, including servers, storage devices, and network resources. Using virtualization, we get several benefits such as raising the resource utilization, sharing the resources, reducing the overall cost, encapsulating and isolating the computing environment of each user in virtual machines, and migrating the virtual instances among different physical hosts.

2.2.1 Hardware virtualization

Hardware virtualization is the creation of virtual resources that act like real ones. The purpose is to separate the compute environment from physical resources and to increase resource utilization. Virtualization is the main building block in cloud computing. Containers and virtual machines are two deployment methods for virtual platforms in cloud computing.

Virtual Machine (VM) is a software that emulates and provides the same functionality of a physical computer. Operating systems can be installed and run on VMs.

The hypervisor, which is also called a Virtual Machine Monitor (*VMM*), allows several virtual machines to share the hardware resources. It is responsible for creating, running virtual machines, and isolating each running instance of a virtual machine from the physical machine.

Containers are isolated user-spaces hosted on operating system instances and may look like real machines. Programs running inside a container cannot see computer resources except those assigned to the container.

Cloud instance is a virtual server. Virtual hardware is implemented by software on top of real hardware. Cloud instances provide several advantages. It is possible to replace the real hardware and move the virtual servers to work on another resource. Adding removing resources using the software is easy which maximizes HW resources utilization. It is possible to expand to multiple machines. It avoids crashes by changing resources. The user does not have to worry about how many servers are needed to run a task.

2.2.2 Application virtualization

In application virtualization, part of the runtime environment is replaced with an application virtualization layer which intercepts all disk operations of the virtualized application and redirects them to a virtualized location, often a single file. Examples include Citrix XenApp, VMware ThinApp, and Microsoft App-V. With application virtualization, it becomes easy to run the application on different computers. The application remains unaware that it accesses a virtual resource instead of a physical one. Some applications cannot be virtualized such as anti-viruses, applications that require a device driver, and applications that use a lot of OS functions.

Service virtualization is the emulation of specific components behavior in component-based applications. Examples include API-driven applications, cloud-based applications, and service-oriented architectures. It is useful for testing and development of software components. It focuses on virtualizing web services.

2.2.3 Desktop virtualization

In desktop virtualization, the operating system (OS) is isolated from the client device of the user. The desktop environment is separated from the physical client device such that no data is saved in the user's device and all components are saved in the data center. The

user basically either logs in to a shared desktop on a remote server which is called session virtualization or connects to a virtual machine hosted in a data center which is called virtual desktop infrastructure (VDI). Some examples include XenDesktop and View.

2.2.4 Network virtualization

In network virtualization, the networking functionalities are implemented on a software-based entity called a virtual network. Network virtualization is useful for software testing since it can simulate the network environment. Some examples include SDN, VMware, NetScaler, and Cisco.

External network virtualization combines or subdivides local area networks (LANs) into virtual networks to improve efficiency. Internal network virtualization uses software to emulate a network. It is useful to isolate applications to separate containers.

The performance of network virtualization suffers when using 10 gigabit/sec networks and above. With these networks, the packet rate might exceed that processing capability. To overcome this limitation, some hardware devices are combined with the software-based network to higher processing performance.

2.2.5 Storage virtualization

In storage virtualization, all storage media are treated as a single pool of storage. It can be either file virtualization which eliminates the dependency between the file name and the file contents locations or block virtualization which introduces logical partitions. In data centers, multiple disk drives are combined and form a disk array and storage virtualization is used to provide them as a storage system. Some examples include NetApp, IBM, Compellent, etc.

2.3 OpenStack: open source cloud computing platform

In this section, we illustrate the OpenStack cloud computing platform as an example to understand the components of cloud computing platforms. OpenStack is an open source software for creating private and public clouds. Unlike most existing cloud computing platforms, OpenStack supports adding and pooling custom hardware and provides infrastructure as a service. It is used in several researches on FPGA integration in cloud computing [9, 10, 11, 12]. In the following, we list some important components of the OpenStack:

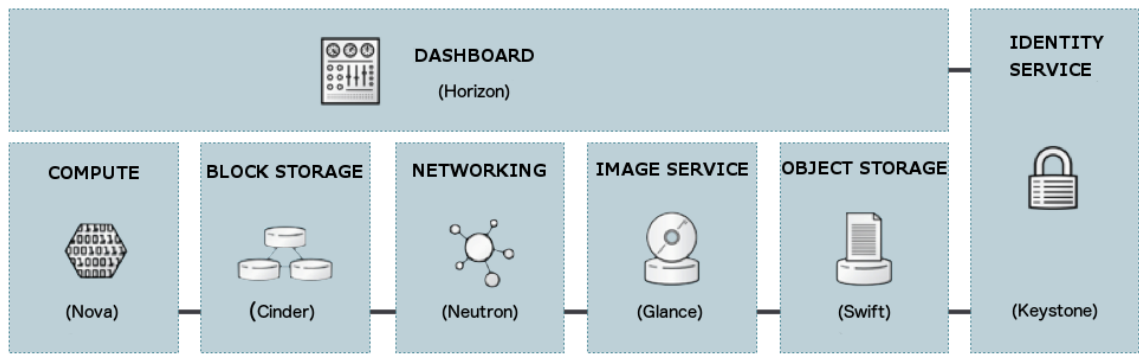


Figure 2.2: OpenStack core components [13].

- *Nova*: The compute nodes controller which provides virtual machines as IaaS. It is written in Python and uses many external libraries. *Python* is an interpreted (i.e. the source code is not compiled and available for modifications) high-level programming language for general-purpose programming. This allows researchers to build a modified version of the Nova that integrates FPGAs as computing resources.
- *Neutron*: The networking manager which is responsible for providing IP addresses and ensuring that the network is not a bottleneck or limiting factor. Users can create their own networks and control traffic. Software-defined networking (SDN) is

supported. It provides general services such as load balancing, firewalls, virtual private networks (VPN) and intrusion detection systems (IDS).

- *Swift*: The redundant storage manager that stores files.
- *Glance*: The image service manager. It can add, delete, share, or duplicate images. It could be used to store backups or to enable VM migration between physical servers at run-time. It enables dynamic optimization of resources and allows performing maintenance.
- *Keystone*: The identity manager that manages the authority and provides a directory of users with the services they can access.
- *Horizon*: The dashboard that provides a graphical interface to access, provision, and automate the deployment of cloud-based resources.

Cloud computing platforms are server-based clouds which depend on CPUs as the main computation device. Even when a non-CPU computing device is offered, it is offered as an I/O device attached to a CPU-based virtual machine. Offering FPGAs as standalone computing devices on cloud computing systems require extending the virtual machine concept to include non-CPU computing devices. With FPGAs, instead of the virtual machine image, that represents the virtual server, we have the bitstream that represents the application hardware. Instead of launching a virtual machine from an image file, we configure an FPGA by a bitstream file. Since OpenStack is an open source cloud computing platform, it provides the flexibility to extend the virtual machine concept to include FPGAs. Its source code can be modified to allow integrating FPGAs as computing resources as seen in several works [9, 10, 12].

CHAPTER 3

Design with FPGAs

Field Programmable Gate Array (FPGA) is a programmable chip that can be configured to be any integrated circuit. FPGA can be reconfigured again and again to hold a different design each time. Early use of FPGA was to prototype and test hardware designs before fabricating them as non-configurable chips (ASIC). Nowadays, FPGA is used in computation either as accelerators attached to a server to accelerate computational intensive tasks or as a standalone device to do real-time computation such as network packet processing.

In this chapter, we introduce FPGA architecture and design flow. We talk about the standard methods of writing hardware designs; the hardware description languages (HDL) and high-level synthesis (HLS) tools. Then, we list strength and weakness FPGA and try to identify the place of FPGA among traditional computing devices; CPUs and GPUs.

3.1 FPGA architecture

Field programmable gate array (FPGA) consists of a large number of reconfigurable blocks with configurable interconnections, Figure 3.1. Almost everything in the FPGA is reconfigurable including the configurable logic blocks (CLBs), the static memories (SRAMs), the digital signal processing units (DSPs), and the look-up tables (LUTs). In addition, the connection between these components is reconfigurable. This makes FPGAs very flexible hardware that can be reconfigured again and again to work as different hardware each time. Putting such hardware resources on the cloud to be available for everyone in an easy way is such a dream that needs serious work.

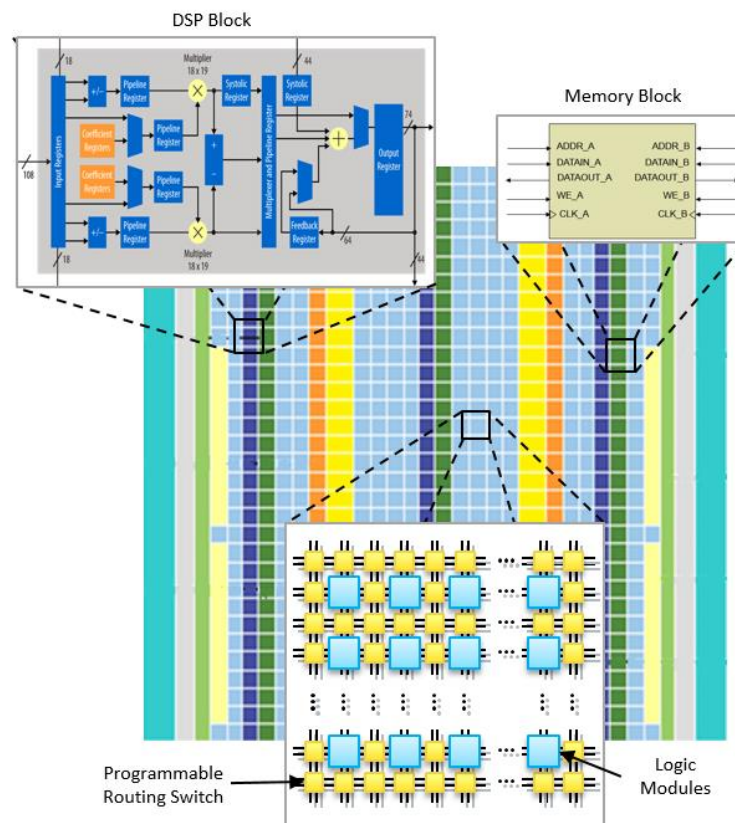


Figure 3.1: FPGA Architecture is a two-dimensional array of reconfigurable resources. Components on FPGA are programmable: control logic blocks (CLBs), SRAMs, DSP blocks, and interconnects [14].

- **Configurable logic block (CLB)** is a configurable block that provides simple logic functions. CLB is the main components that exist in every FPGA. Each CLB contains several slices. Each slice contains several Flip-Flops (one-bit memory) and Look up table (LUT).
- **Look up table (LUT)** could be seen as a memory that stores the truth table of a logic function, Figure 3.2. The LUT inputs work as an address of that memory. This way it can represent any logic function. Recent FPGAs contains 4-, 5- or 6-input LUTs.

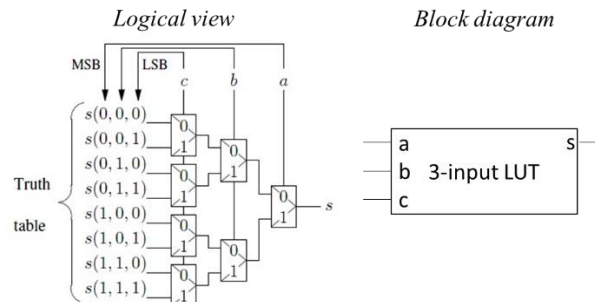


Figure 3.2: 3-input LUT implementations

- **Block RAM (BRAM)** is a memory block that can be used to store data words in a specific address. Read and write to BRAM consumes only one or two cycles. In general, BRAM has three inputs which are address, write enable and input data and has one output for reading data. BRAM can be a dual port. In this case, its input pins and output pins are doubled.
- **Digital Signal Processor (DSP)** is a configurable block that contains multipliers, preadders, adders, subtractors, accumulators, coefficient register storage, and a summation unit. The DSP is commonly used to implement floating point operations.
- **Clock generator and buffers** provide clock distribution system with a configurable frequency. Some clock buffers are controllable and allow stalling the clock signal to freeze the design that operates by this clock.

FPGA is configured by a stream of bits called the configuration bitstreams. The bitstream contains all needed information about FPGA components such as their type, locations, and their initial values. The configuration process is done in seconds while generating this bitstream could take minutes. The bitstream represents the image of the application hardware. Modern FPGAs supports *dynamic partial reconfiguration* (DPR) which is the ability to reconfigure part of FPGA at run-time without disturbing other parts. The bitstream size is proportional to the reconfiguration region size not to the application hardware size.

3.2 Design flow

The traditional method of designing application hardware is to write a code using the Hardware Description Languages (HDL) such as Verilog and VHDL. Then, the code is synthesized by specific synthesis tools provided by the FPGA vendor and then, its bitstream is generated for that FPGA. The compilation process for HDL to bitstream consists mainly of the following steps:

- **Synthesize:** The HDL file is compiled to the netlist. The netlist is a list of general primitive components (LUTs, FFs, BRAMs, etc.) and their interconnections.
- **Map:** The primitive components in the netlist is mapped to actual components of the specified FPGA device. This is a time-consuming process.
- **Place and route:** The components are placed and connected. Heuristic algorithms are used to find the best places and the best routes to match the area and performance constraints. It is the most time-consuming process and heuristic algorithms are extensively used.
- **Bitstream generation:** The final configuration bitstream is generated.

The compiling process takes minutes and hours (e.g. 30 minutes). Large designs in a small area take longer than small designs in a large area. Modern FPGAs takes longer compilation time because they have more resources.

3.3 Partial Reconfiguration

Modern FPGAs support partial reconfiguration which allows configuring a portion of the FPGA. Dynamic partial reconfiguration (DPR) allows configuring part of the FPGA while other designs on other parts on the FPGA is working. The partial bitstream has a smaller size and it is only applied to that partial region.

FPGA can be configured externally through a serial interface such as the JTAG interface or internally through internal configuration module provided by the FPGA manufacturer. Xilinx introduces the Internal Configuration Access Port (ICAP) that can do partial configuration. It introduces the PlanAhead tool that supports the generation of partial bitstream. Altera introduces the configuration via a protocol (CvP) that allows configuring FPGA through the PCIe interface. They introduce the Intel® Quartus® Prime Tool that enables the generation of the partial bitstream for their FPGAs.

Partial reconfiguration faces several limitations and needs a careful understanding of the device architecture and its configuration memory. The following examples illustrate how does it affect the reconfiguration region size and shape. In Xilinx Virtex II and Virtex III devices, the configuration memory is organized as a 2-D bit array in which each column represents one frame [15]. A frame is an atomic unit of configuration. The frame contains configuration bits that belong to several physical resources that share the same column. Those resources are configured using several frames. Using Xilinx Virtex II and Virtex III

devices, it is difficult to have two partially reconfigurable regions that share several columns. We may need to extend each design for the full height of the device.

In Virtex 6 devices, the configuration memory is organized as a 2-D bit array of frames [16]. Each row of the array represents a clock region. Each column of the array represents contains a single type of FPGA primitive. The CLB frame contains 40 CLBs. the DSP frame contains 8 DSPs. The BRAM frame contains 8 18Kbit Block RAMs.

In addition to these limitations, the interfacing between the partially reconfigurable region and the static region needs special considerations. For example, Xilinx FPGAs requires all I/O of the partial region to be registered. Those registers should be disabled while doing the partial reconfiguration. In a recent version, Xilinx Vivado can automatically add partition pins [17].

For partial reconfigurations, Intel introduces the secure device manager (SDM) [18] for Stratix 10 and the PR control block IP core [19] for Arria 10 FPGAs. The PR IP core contains three controllers; Control Block Interface Controller, Freeze/Unfreeze Controller, and the Data Source Controller. The Freeze/Unfreeze controller provides a signal that is used to freeze the interface between the partial region and the static logic during the reconfiguration process. Intel uses the term “PR persona” is used to indicates a partial bitstream [20]. As explained in Intel documentation [21], partial reconfiguration may corrupt the contents of some BRAM blocks in the static region if careful floor planning is not taken.

Finally, not all resources are dynamically reconfigurable. Clock buffers, IOBs, and transceivers are usually non-dynamically reconfigurable resources. The internal

configuration of DSP blocks and the initial values of block rams cannot be dynamically changed.

3.4 High-level synthesis

Although hardware description languages (HDL) is the natural method writing hardware codes, HDL describes the compute machine architecture not the computing task algorithm. High-level synthesis (HLS) tools provide higher abstraction because it uses C-like constructs. Designing hardware that runs on FPGA is done by writing code in hardware description languages (HDLs) which are mainly VHDL and Verilog. HDLs provide a very low abstraction level analogous to assembly in software design. It requires developers with a strong background on designing digital circuit components such as registers, counters, control units, ALUs, data path, state machines, etc. It also requires a deep understanding of digital circuit terminologies such as clock, synchronous and asynchronous signals, inter-domain communication, hand-shaking protocols, etc. In addition, the designer should understand the FPGA components to be able to utilize its components and debug his code.

With the increasing trend of using FPGAs to do computations, extensive researches have been done on compiling algorithms written on high-level programming languages (HLL) to hardware description languages (HDL). A lot of HLL to HDL compilers are invented and researched. A lot of problems raised because HLLs are built on top of the Von-Neumann architecture and sequential execution with memory hierarchy, not the inherently parallel HW execution model. HLL constructs such as recursion and dynamic memory allocation cannot be supported on FPGAs unless a soft processor (i.e. a processor built using FPGA's configurable resources) is used. However, using soft processors results in very poor performance. Compiler directives are suggested to indicate blocks that can be

parallelized just like those that target GPUs or multi-processing systems. Almost all HLL to HDL academic researches ends up to a commercial HLS tool. HLS tools are the state of the art of HLL to HDL researches. An HLS tool accepts as input a C-like code that describes the user application and produces as output a hardware description written in HDL or register-transfer level (RTL).

HLS provides higher abstraction since the application can be expressed in fewer lines of code. It also decreases the designer design time. High-level synthesis (HLS) tools allow designers to efficiently explore the design space trading off performance for area and/or power. However, it increases the compilation time since the total design time is increased. It produces less efficient circuits compared to handwritten HDL code. HLS tools are vendor specific. HLS tools still require hardware design background that is beyond typical application developers. Some HLS tools such as BlueSpec, SystemVerilog, and SystemC may facilitate hardware-designers life by hiding some low-level details such as the clock signals but they still need the user to be aware of the underlying hardware components. Some other HLS tools are suitable only to design accelerators for specific application domains. There are few open-source HLS tools such as Leg- and Bambu [22].

- *OpenCL* [23] is an extension to C/C++ language that allows writing kernels to be executed in accelerators such as GPUs and FPGAs. The user writes a C program and defines several kernels. The program is implemented in a CPU while the kernel is run on the accelerator. Intel introduces the Intel SDK for OpenCL which is a compiler from OpenCL to Altera FPGAs. Xilinx introduces the SDAccel [24] which is a compiler from OpenCL to Xilinx FPGAs. SDAccel is used by Amazon EC2 F1 instances [7].

- *Vivado HLS* [25] is a compiler that targets only Xilinx FPGAs. It accepts C/C++ and SystemC programs and supports C++ classes, templates, functions and operator overloading. It also supports converting OpenCL kernels to IPs.
- *Legup* [26] is an open-source academic HLS tool. It accepts a C code, debug it and run it in software. Then, the HDL code is generated for the function that is declared as a hardware core. It allows the user to define FIFO-based inputs to a function and provide a library for reading from and writing to the FIFO. The last version 5.1 is a commercial version and supports automatic pipelining for specified function or loop. It also supports parallel threads written by pThreads [27].

3.5 FPGA strengths and weaknesses

Compared to other processing units such as CPUs and GPUs, FPGAs achieve better in terms of energy consumption per operation. In addition, due to the large amount of computing logic resources, FPGAs expose better performance than CPUs for applications that contain parallel tasks. CPUs runs parallel threads by switching between threads, the parallelism in FPGAs is true parallelism (hardware parallelism). They represent an excellent option for accelerating and for real-time control systems. However, when it comes to sequential tasks, FPGA exposes poor performance compared to CPU which run on gigahertz while FPGA runs at 10s or 100s megahertz in the best case. Usually, an application contains both. It contains a parallel part and a sequential part.

Figure 3.3 shows computation models comparison among CPU, GPU, and FPGA. CPU is based on the von Neumann computing model. GPU is based on vector processing model since it has several parallel processing cores. FPGA uses a chain of dedicated processing

elements instead of performing computation on one ALU. It can explore parallelism in different ways and only if there are enough resources.

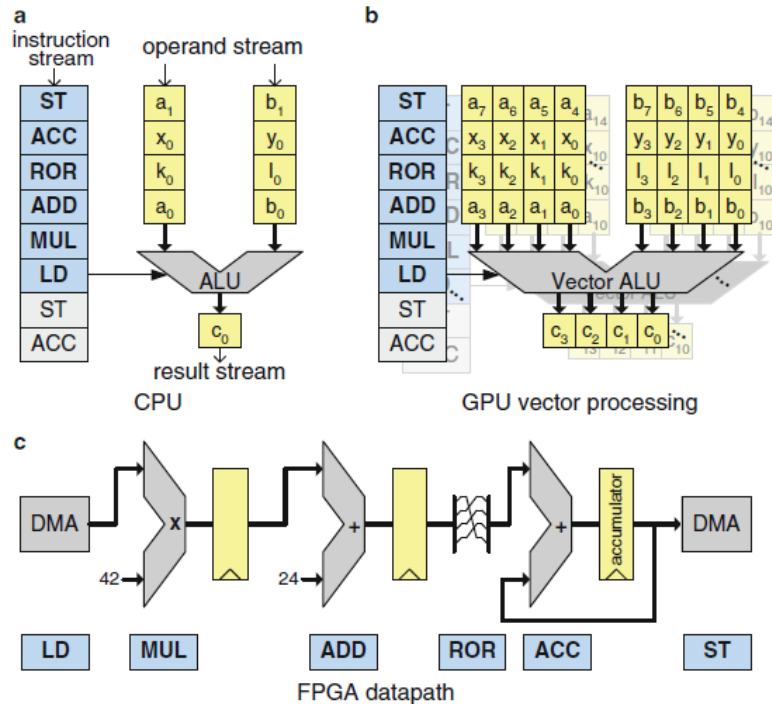


Figure 3.3: Compute paradigms comparison [28]. a) CPU represents the Von Neumann model. b) GPU represents the vector processing model. c) FPGA represents spatial computing.

To show the strength and the weakness of the FPGA, we compare different computing devices, CPU, GPU, and FPGA. This helps to decide how and when to use FPGA for computation and what computation model is best suited to FPGA. Many advantages motivate using FPGA in computation as an accelerator or as a standalone computing machine:

- The latency of FPGA is better than GPU. FPGA is more suitable for real-time computation. It provides deterministic time with accuracy reaches to nanoseconds.
- FPGA can be programmed to deal with any type of interfaces, unlike GPUs which are restricted to the PCIe interface.
- Low power is another factor that makes FPGA a good computing machine.

FPGAs suffer from many limitations that prevent them from being widely used in computations:

- FPGA are *expensive* compared to other computing devices. Their design flow tools licenses are expensive too. The design time of FPGA is the worst.
- Compiling an HDL code to a bitstream could take minutes and hours. Hardware simulation and debugging is also a very *time-consuming process*. In contrast, the compilation time for GPUs and CPU is almost instant. Their debugging time is much better than that for FPGA.
- Application *size* in FPGA is much smaller than that in GPU and CPU because FPGA does not use instruction memory, instead, it built special hardware for the application. The performance versus area tradeoff makes smaller applications performs better on FPGA compared with other applications.
- Software programming provides high flexibility while FPGA is the worst. GPU is slightly better in applications that need floating point operations. FPGA uses DSP blocks to do floating point operations, but it has a limited number of DSPs.

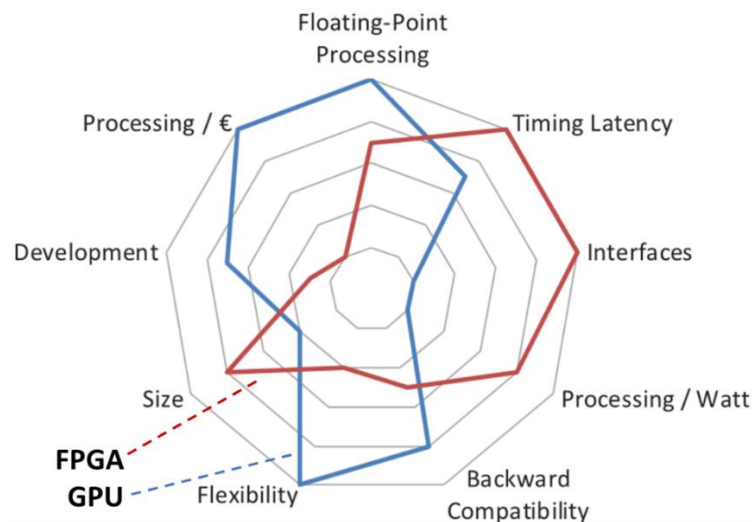


Figure 3.4: GPU vs FPGA qualitative comparison [5].

CHAPTER 4

Literature Review

The techniques and definitions of FPGA virtualization are changing over time. Two common approaches can be seen in recent works for virtualizing FPGAs; FPGA overlays [29] and virtualization by adding abstraction layer (AL) that abstracts the FPGA interfaces. In this chapter, we explore the FPGA virtualization works. Then, we explore the other works on integrating FPGA in data centers and clouds.

4.1 Overlay architecture

Overlay architectures are to build virtual FPGAs on top of physical FPGAs to reduce vendor dependencies [29]. The overlay works as an intermediate layer between the user hardware and the vendor FPGA. First, the several instances of the same overlay are synthesized and implemented on different FPGAs types. Then, the user hardware is translated only one time to the overlay and become able to run on all FPGAs. Overlays can exhibit features independent from the host FPGA such as allowing context switching by reading configuration bitstreams as well as the contents of the registers. Overlays have three main advantages over physical FPGAs:

- *Portability*: The user application become completely independent from the physical FPGAs and their vendors. One bitstream represents the application for devices.
- *Maintainability*: Upgrading physical FPGAs and replacing old one become smooth.
- *Migration ability*: Context switching and time-multiplexing several applications become possible. It is possible to migrate a running application from a physical location to another. The demonstration in [30] shows live migration between two nodes of a cluster of heterogeneous FPGAs.

Overlay architectures can be fine-grained [31, 32] or coarse-grained [33, 34, 35]. In fine-grained overlays, FPGA components such as LUTs, CLBs, interconnects, etc. are modeled in HDL, synthesized and a bitstream is generated for the new configurable virtual FPGA. The enable signals in physical registers become a virtual clock. Fine-grained overlay involves *high area overhead* and causes *performance penalty* because it lowers the operating frequency. *Productivity is too low* in fine-grained overlay because its input is HDL code which needs to be synthesized, placed and routed using similar tools of normal FPGAs. Coarse-grained overlays contain larger and *more abstract components* such as FFT, SQRT, adder, multiplier, etc. that target a smaller number of applications. The coarse-grained overlay is less general purpose than fine-grained ones. The coarser the cells are the more domain-specific we get. Restricting it to *specific application domains* is the main disadvantage. The coarse-grained overlay does not suffer from area and performance overhead like fine-grained overlays. Productivity is high, the instant compilation is possible and therefore their applications are easy to debug. Configuration time is also fast because their configuration bitstreams are small. They are easy to be used by a software

programmer. The programmer introduces the HLL code, its control flow graph (CFG) is extracted, and finally, the interconnection configuration is produced.

Although overlay architectures provide portability which fits well with virtualization requirements, the cost in area and the performance overhead in fine-grained overlays make them not practical for using FPGAs in computations. In coarse-grained overlays, it is difficult to have general FPGA virtualization platform since coarse-grained overlays are domain-specific. Therefore, works on integrating FPGA in cloud and data center does not use overlays as virtualization techniques, instead, they use virtualization by adding an abstraction layer (AL).

4.2 Virtualization using abstraction layer (AL)

In this type of virtualization, an abstraction layer (AL) is created which contains a collection of common hardware components that are needed by all applications. A software library is also created to provide standard API functions that interact with the abstraction layer. The hardware components of the abstraction layer are:

- *Communication controller* such as PCIe endpoint, Ethernet controller or off-chip memory controller.
- *Routing*: It is included when the platform supports several vFPGAs per a physical FPGA.
- *Reconfiguration management*: It is included if dynamic reconfiguration of vFPGAs is supported.
- *Clocking management*: It is included if different clock domains are needed.

- *Soft processor*: In some works, a soft processor is included as a reconfiguration manager. Other works used the soft processor to control vFPGA and the application run.
- *Security blocks*: Some platforms add security modules at the abstraction layer level.
- *Registers*: It collects information and status or store configurations that control the abstraction layer and vFPGAs

Each physical FPGA is divided into two parts; the abstraction layer (1% up to 25% of the physical FPGA area) and one or more vFPGA regions to hold the applications. The size of the abstraction layer differs from work to another and depends on what components are included/excluded in it. Each research refer to the abstraction layer using different terminology such as static logic [36, 9], RC2F [37], service layer [10], vendor logic [12, 38], network service layer [39], FPGA hypervisor [11, 40] and shell [7, 41]. In some researches, the vFPGA region is suggested to be a dynamically reconfigurable region [9, 10, 12, 37] while in other works [7, 42] the user design is combined with the abstraction layer and compiled together to form one full bitstream.

The abstraction layer has a hardware side as explained above and has a software side. Software libraries are introduced in several platforms for PCI-attached FPGAs [43, 37, 44, 36, 45]. Knodel [37] for example, introduces the RC3E FPGA hypervisor that provides functions for device control, vFPGA control, data flow control that interacts with the off-chip memory, load bitstream, get status and set configurations. The provided API functions by different works can be categorized as follows:

- API functions for device info and existence queries; *get_device_info()*

- API functions for accessing vFPGA, FPGA or an individual channel on FPGA through *send()* and *receive()* API functions.
- API functions for reading or writing to registers in the FPGA.
- API functions for resetting FPGA and the transfers on all channels.

There are important researches that address some virtualization properties. Merging several vFPGA regions to form one large region is discussed in [46]. This allows having vFPGA of different sizes. Migrating a design from one vFPGA to another is discussed in [47] in which the measured migration time is around one second.

4.2.1 Interfacing the abstraction layer (AL) with vFPGAs

Platforms that don't support dynamic partial reconfigurations (DPR) for vFPGAs don't require high abstraction in the vFPGA interface. This is because the application and the abstraction layer are merged and compiled together as one bitstream. However, if DPR is supported, the vFPGA interface should be a fixed interface that fits for all applications.

The AXI [48] interface is commonly used for vFPGA interfacing in several works. It is a handshaking interface that allows both sides of the link to stall (stop) data movement. It provides efficient data interchange and can offer one data element per cycle. FIFO-based interfaces are also used in [37, 40, 12, 39, 44]. The empty and full signals of the FIFO allow both sides to know when the other side is busy. Asynchronous FIFO is used when the application works on a different clock than the abstraction layer. Amazon EC2 F1 instance [7, 49] uses three AXI-4 interfaces, three AXI-Lite interfaces, and some generic signals such as; clock, reset, status, etc. In OpenCL based platforms and other GPU-like platforms, vFPGA interface act as a DMA controller since the vFPGA interface is restricted to the off-chip memory [44, 38].

Some works [39, 44] indicate that some standard logic might be needed inside the vFPGA to abstract its interface. The vFPGA interface in DyRACT [44] consists of streaming channels and FIFOs inside the vFPGA to be added by each application. The application is clocked using the control logic's frequency. Another version of this work [36] uses asynchronous FIFO-based SDRAM interfaces. However, they did not introduce a general abstraction method instead, they left this to the application designer to build a specific wrapper for the application. Asiatici et al. [50, 51] suggest adding onboard processor or soft processor to the shell for orchestrating the hardware accelerator execution. A manager executing parallel constructs on the processor manages the FPGA resources and communicates with the host over the PCIe.

4.2.2 OpenCL and the CPU-FPGAs interface abstraction

OpenCL defines a mechanism [52] for CPU-FPGA communication and supports API functions for querying devices info, FPGA partitioning, and command queues (such as a memory write and read or executing a kernel). Just like OpenCL for GPUs, the application is written in HLL language where the compute-intensive functions are defined as kernels to be accelerated on PCI-attached FPGA. The compiler compiles the software part and the hardware and manages the interfacing between them. This helps the designer to focus only on his application logic.

Xilinx and Intel developed their own OpenCL compilers for FPGA devices [53, 54]. The compiler compiles the kernels into hardware on vFPGA and integrates the abstraction layer with them according to a predefined architecture explained in their documentation [55, 14]. The Xilinx OpenCL compiler is used by several works [7, 11].

Amazon announced EC2 F1 [7] which is a cloud compute instance that provides FPGA infrastructure as a service (IaaS). It uses FPGAs and design tools from Xilinx. Amazon EC2 F1 provides two methods for building on-cloud hardware. The *first approach* is to use the OpenCL development environment to build heterogeneous applications that run on CPU and accelerated on FPGA. They have used the Xilinx SDAccel tool and the user should obtain a license from Xilinx and install it to use these tools. The *second approach* is to install and use the Vivado IP Integrator (IPI) provided by Xilinx. In this approach, the user is free to build cores that can be accessed over PCI. To abstract the FPGA interface, Amazon EC2 F1 introduces the Hardware Development Kit (HDK) library. The HDK integrates the static logic (shell) with the Custom Logic (CL) provided by the user. The shell interface at the hardware side is connected to a DRAM controller and/or PCI express controller. The shell interface at the custom logic (CL) side provides several AXI interfaces. There are three AXI-4 interfaces, three AXI-Lite interfaces, and some generic signals such as clock, reset, status, etc. The Shell and CL are synchronous to one clock with a maximum frequency of 250MHz. There are seven more clock signals. The developer can select among a set of available frequencies provided in the clock recipe table. There are four DRAM interfaces. One of the four DRAM interface controllers are implemented in the Shell, and three are implemented in the Custom Logic (CL) [55].

Amazon EC2 F1 introduces the Amazon FPGA Image (AFI) management tools [56] to manage FPGA images (bitstreams) on the cloud. It includes functions for listing available FPGA slots, getting image status, loading image, clearing image, start virtual JTAG to debug the design, get/set LEDs and switches.

Intel acquired Altera (a manufacturer of FPGAs) in 2015. Intel introduces FPGA as PCIe attached devices just like GPU devices. The OpenCL language is used to write software that runs on CPU and define some kernels that run on FPGA. Abstracting the FPGA interface in Intel FPGA is done by using the off-chip DDRAM as a shared memory between CPU and FPGA. First, the data are sent from the system main memory over the PCI to the FPGA board off-chip DDRAM. Second, the accelerator is invoked to start. Third, the accelerator does computations on the data and store results on the DDRAM. Forth, when the accelerator finishes it interrupts the CPU. Finally, the results are moved back to the system main memory at the CPU side [54].

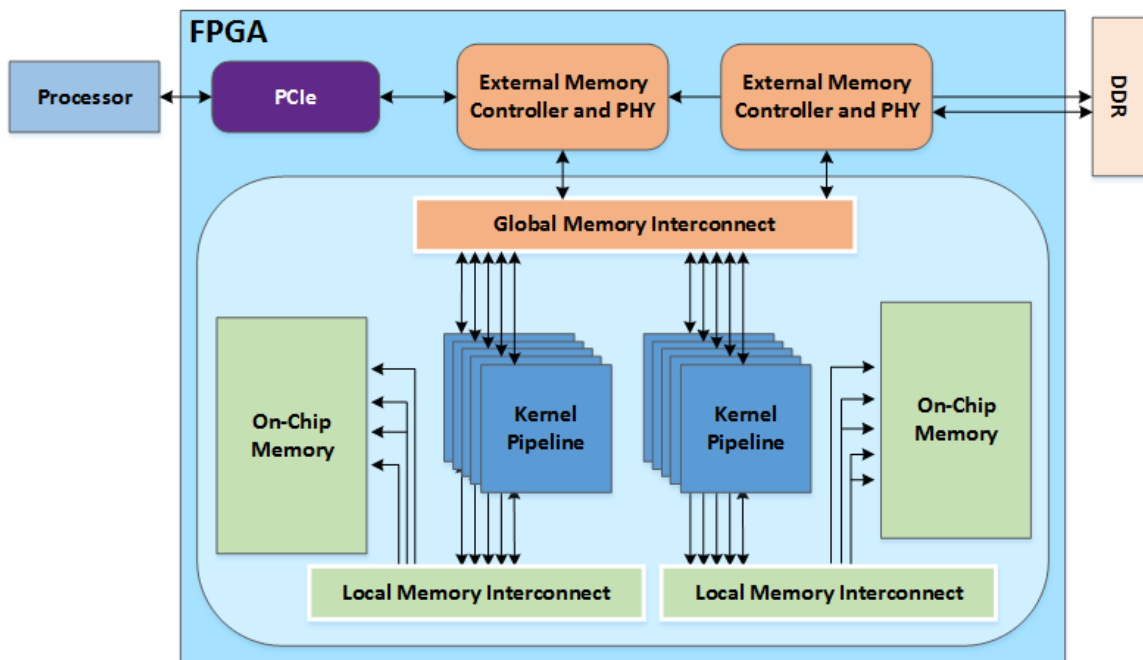


Figure 4.1: The architecture and design components in Intel OpenCL for FPGAs [14].

4.3 FPGA attachment interface

FPGA-based compute node properties are severely affected by the style of physically attaching FPGAs to the system. PCIe and Ethernet interfaces are the most commonly used approach because of their high bandwidth.

With PCIe interface, the FPGA is physically coupled to a CPU-based server. Integrating FPGAs on cloud computing platforms is done by introducing compute nodes or virtual machines with PCIe-attached FPGAs [57, 37, 36, 10, 7]. In this case, the VM hypervisor is slightly modified to launch VM requests in machines that have PCI-attached FPGA [11]. Wang et al. [58] show a different method in which PCI-attached FPGA can be common among several servers. They used the Xen virtual machine monitor (VMM) to provide FPGA access to all servers on the network and manage the PCIe traffic between servers and hardware. In [59, 60] a technique is proposed to make a single PCIe-attached FPGA appears as several separate coprocessors to multiple VMs running on the same host node. Taking advantage of the PCIe Single-root I/O virtualization capabilities, each vFPGA appear as a separate PCI-attached FPGA and each VM got exclusive access to its “own” vFPGA.

The tight coupling between FPGAs and CPUs in compute nodes leads to several limitations:

- The number of FPGAs in a DC is limited to the number of CPU nodes and PCIe slots per node,
- FPGAs cannot be used independently from the CPU node they are attached to; i.e. CPUs must explicitly send/receive data and instructions to the FPGAs wasting both CPU's and FPGA's cycles,
- In a cloud setting, customers actually instantiate two compute instances (one on a CPU and another on an FPGA),

- Aggregation of several FPGAs to implement a large application becomes difficult and inefficient as the data traffic between these FPGAs must go through the nodes (i.e. no direct communication between FPGAs).

Weerasinghe [12, 38, 39] proposed connecting FPGAs to the data center network as standalone computing devices. In [61], network-attached FPGAs are configured to one of three block types; compute core for general-purpose computing, memory block, or hardware acceleration. These types allow users to choose the most suitable architecture and memory management for their applications.

In Microsoft catapult [6] both interfaces are used. FPGAs are connected using a secondary network while they are still PCI-attached devices. A large application is implemented on a chain of FPGAs such that each FPGA implements one phase of the application. Each FPGA contains the abstraction layer and one application phase. Then, several copies of the same application are launched on several to raise the throughput.

4.4 FPGA in the cloud and data center

Cloud computing systems use CPU-based servers to do computations. Integrating FPGA as a computing resource is not supported directly. Large companies add modifications to their own cloud systems to support their FPGAs. Microsoft Catapult [41] is hosted on the Microsoft Azure cloud computing platform. Amazon Cloud hosts the EC2 F1 instance [7]. Several academic works [9, 10, 11, 12] suggest modifications in the OpenStack cloud computing system to enable integrating FPGAs as computing resources. In OpenStack, FPGA could be integrated to accelerate the data center functionalities but not provided as cloud servers. Microsoft uses FPGAs to accelerates Bing search [6]. It also uses SDN on

FPGAs to accelerate its networking operations [2]. The user is not aware of the FPGAs existence.

Byma, et al. [9] introduced a method for virtualizing FPGAs to enable their integration as standalone compute nodes (i.e. not coupled to a CPU). This is the first work, as far as we know, that introduces virtualizing FPGAs using abstraction layer method. The abstraction layer includes ethernet controller, a soft processor, and a memory controller. vFPGAs are introduced as Infrastructure-as-a-Service (IaaS) and the OpenStack cloud system is adapted to manage vFPGA resources. Configuration is done using a JTAG interface and a UART is used to configure network address registers. This meant that the FPGA still requires to be attached to a server to be configured by bitstreams or to configure its network address registers. The cost of integrating FPGAs into data center increases due to excess cabling.

Tarafdar et al. [11], suggest a platform for attaching FPGAs to virtual machines (VMs). The platform assumes that the VM is running on a server that has PCI-attached FPGA. The Xilinx SDAccel Platform [24] is used to provide the abstraction layer and the software drivers. It is deployed using the OpenStack cloud system and provided as Infrastructure-as-a-Service (IaaS). OpenCL allows the user to write his application with kernels, compile it, configure it and abstract the VM-FPGA interface, enabling the user to focus on the application logic.

Knodel et al. [37] introduced RC3E as a framework to integrate FPGAs in the cloud. In their framework, each physical node includes a server with several FPGAs connected to it through PCIe. They introduced their abstraction layer which is called the RC2F. It consists of PCIe endpoint, clock management, and control and reconfiguration manager. They

introduce API functions like those provided by OpenCL. Their platform allows introducing accelerator as a service or the vFPGA region as a service.

Kidane et al. [40] adapted the network-on-chip (NoC) architecture to build a virtualization platform. The idea is to map each processing element (PE) in NoC to a dynamically reconfigurable region. The switches (routers), links and network interfaces become the abstraction layer. The platform introduces the reconfigurable region as a service or the reconfigurable IP as a service.

Chen et al. [10] did major effort to integrate FPGAs as accelerators. Multiple virtual FPGAs were tightly coupled with one CPU through a PCIe interface. Multiple processes can be scheduled on the same virtual FPGA (through re-configuration). The user configures a virtual FPGA with a customized accelerator with a custom communication interface. The system can context-switch the same accelerator among users. There is no virtualization at the level of interconnection. Another drawback is that the PCIe interface becomes a bottleneck and its bandwidth should be balanced as discussed in their paper.

4.5 ASIC Clouds

ASIC Clouds [62] suggest fabricating several instances of the same accelerator with routing and interconnection on an ASIC chip and having several ASIC chips on boards and racks. There are on-PCB network and a control plane that interpret incoming packets from the on-PCB network and schedules computation and data onto replicated compute accelerators. Unlike FPGA-based accelerators, the designed hardware in ASIC chip can never be modified but it provides the highest possible performance with low power. They showed that ASIC cloud optimizes the total cost of ownership (TCO) to 2-3 orders of magnitude better than CPU and GPU on four case studies; the Bitcoin mining ASIC

Clouds, YouTube-style video transcoding ASIC Cloud, a Litecoin ASIC Cloud, and a Convolutional Neural Network ASIC Cloud.

4.6 Summary

Table 4-1 below shows a summary of notable platforms of FPGA-based processing for clouds or datacenters. The ASIC cloud is included in the comparison, although it is not reconfigurable hardware, because it provides ASIC-Based custom computing machines for the cloud.

The configuration/attachment refers to how the FPGA is attached to the cloud (or datacenter) and how it is used. FPGA-based accelerators are attached to a host CPU via the PCIe bus and cannot be used independently from the host (i.e. in a cloud environment, the user need to instantiate two compute instances). Depending on their shells, FPGAs attached to the data center's network can be used on their own (i.e. standalone) or still need to work in tandem with a host CPU that runs the main application and calls FPGA acceleration functions over the Ethernet. The latter option also requires two compute instances. Network-attached standalone FPGAs act as servers (i.e. can be used by multiple users/applications). Microsoft's Catapult provides all types of attachments and configuration at a staggering logic cost [6].

The clustering column shows if several FPGAs can be connected directly to run large applications without having the data going through CPU nodes. JetStream is the only PCI-attached FPGA that allows a vFPGA-to-vFPGA connection.

The IF (Interface) abstraction column reflects the level of abstraction for the application interface. Our proposed platform can receive data in their original format, so it provides

full abstraction. Medium abstraction is provided by FIFO interfaces. A platform with a low abstraction is one that requires users to adapt their design to its fixed interface. Platforms that require the users to develop custom interfaces have no IF abstraction at all.

The DDRx column specifies if a platform's shell has an off-chip memory management interface. We have not opted for this option as it increases the static logic area significantly and it is not crucial for streamed applications.

IBM's network-attached FPGAs [39] is the closest work to our work. Its interface abstraction is medium because it introduces fixed FIFO-based interfaces and the data formatting, and the computation control is completely left for the application's designer to design. The table shows that standalone CCM with abstracted data interface is not introduced by other FPGA virtualization platforms. It also shows that our proposed platform provides ultimate flexibility with relatively low overhead.

Our platform introduces network-attached solution and support in which several virtual FPGAs can be chained to perform computation phases for large applications. The interface abstraction is complete such that the user sends data with no controls. We do not have and off-chip memory support because we target streaming applications.

OpenCL-based platforms completely abstract the FPGA interface allowing the user to focus on his applications. The computing model of OpenCL assumes that FPGA will be attached to CPU and used only as accelerators that are controlled by a running software application. The DDRAM is used to store data before computation and store the result after computation is done. When integrated into the cloud, OpenCL-based platforms are provided within a virtual machine. Therefore, it is provided as a platform as a service

(PaaS). Our platform completely abstracts the FPGA interface allowing the user to focus on his applications. It provides standalone FPGA that is not attached to CPU. It executes the application and provides its service without external control. There is no DDRAM to buffer data or store results. The computing model is streaming compute model. When integrated into the cloud, our platform introduces the application on FPGA as a service which is a form of software as a service (SaaS).

Table 4-1: List of notable platforms of FPGA-based processing for clouds or datacenters.

Work		Configuration/ Attachment to a host	Clustering	Interface Abstraction	DDRx
[37]	RC3E	PCIe-attached		Low	
[44]	DyRACT	PCIe-attached		Low	✓
[36]	Fahmy	PCIe-attached		Low	✓
[50, 51]	Asiatici	PCIe-attached		Low	✓
[12]	Hyperscale	Ethernet-attached	✓	Medium	✓
[38]	IBM's Disaggregated	Standalone, Network attached		Low	✓
[39]	IBM's Net-attached	Standalone, Network attached	✓	Medium	
[11]	Tarafdar	PCIe or Ethernet-attached	✓	None	✓
[7]	Amazon	PCIe-attached		None	✓
[6]	MS Catapult	PCIe-attached + Network attachment + FPGA Network	✓	None	✓
[63]	Byma	Standalone, Network attached		None	✓
[10]	Chen	PCIe-attached		Low	✓
[43]	RIFFA	PCIe-attached		Medium	
[62]	ASIC Clouds	PCIe-attached + Standalone Network attachment		Medium	✓
[64]	JetStream	PCIe-attached	✓	Medium	

CHAPTER 5

Overview of the Cloud-Based FPGA Custom Computing Machines Platform

In this chapter, we are explaining our proposed cloud system which offers virtual FPGAs (vFPGAs) as custom computing machines (CCMs). The focus of this chapter is on general concepts. Implementation details will be explained in next chapters. The full platform is not implemented as it is explained. For example, we have implemented a light version of the network controller. We did not implement the full network controller stack.

The CCM is a network-attached vFPGA configured with hardware that does computations on streamed data. The CCM can be accessed only through a socket interface. The CCM is highly abstracted such that it works directly with streamed data without data reformatting or any additional control information. The data reformatting information and the protocol of applying data to the hardware are completely included within the CCM. The streamed results produced by a CCM are also ready to be consumed by another CCM or software function.

Unlike other works [9, 10, 11, 12] that integrate FPGA in the cloud by doing a modification on existing cloud system components, we are providing standalone FPGA cloud system that can be integrated to other cloud systems without modifications in their component. The FPGA hypervisor is a standalone entity that manages FPGA resources. The computing nodes on our platform are the CCMs which can be connected directly to the cloud network. The FPGA hypervisor contacts the image management to fetch CCM images from the storage. Image management fetches and stores bitstreams using the same functions provided by the cloud system to access the cloud storage.

5.1 FPGA Virtualization

The FPGA virtualization platform consists of several abstraction layers shown in Figure 5.1. In the following, we list and explain those abstraction layers. The implementation and the hardware details of the virtualization platform will be explained in the next chapter.

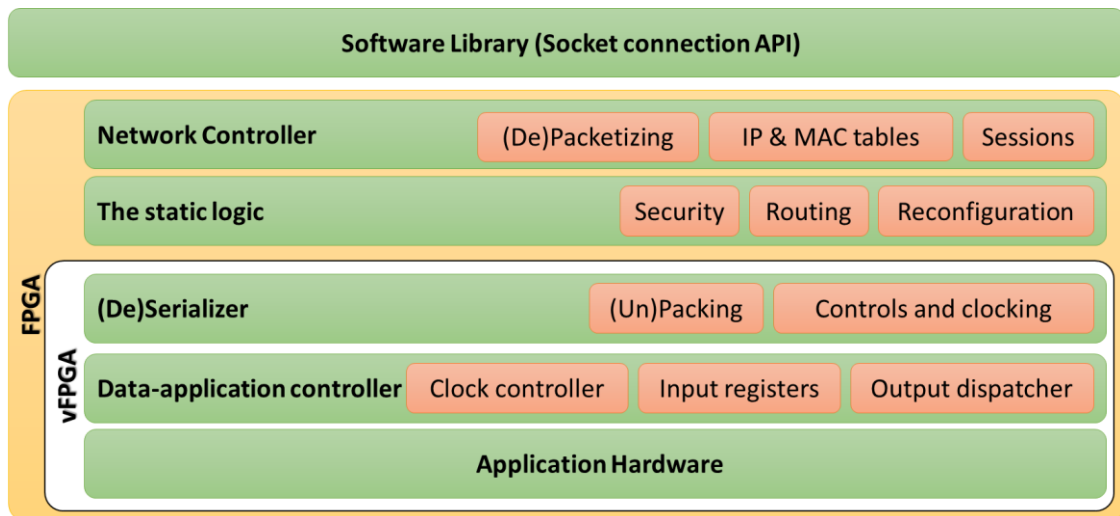


Figure 5.1: FPGA virtualization is based on several abstraction layers.

- *The software library* is the highest abstraction layer and provides several functions for launching, using, releasing CCMs and other management functions.

- *The network controller* handles physical-level connections and networking protocols and establishes TCP network sessions between the users and their applications in the virtual FPGAs. It assigns one MAC address and one IP address to the static logic as well as each vFPGA and makes each of them appear on the network as a standalone network-attached device.
- *The static logic* provides several functionalities. It manages secure data traffic with each CCM. It also reconfigures a vFPGA by a received bitstream.
- *The serializer* receives the incoming data from the network controller through the static logic. The packing/unpacking units change the data width and the serializer reformats the data, adds controls and generates timing information according to the specifications of the application hardware stored within it.
- *The deserializer* removes controls from the results and produces results in standards data format. Then, the packing/unpacking units change the data width to match the fixed-width interface between the static logic and the vFPGA.
- *The application controller* receives the data with timing information and applies them to the application hardware. It controls clocking the application hardware according to data arrival. It gates the input clock to the application hardware. To gate a clock, a controllable clock buffer is used which is a primitive resource in most FPGA devices. The application controller also contains information about which output should be produced and how many clock cycles should be applied for each datum.
- *The application hardware* can be any hardware design with an arbitrary interface. The automatic wrapper generator generates a specific wrapper for each application allowing to fit in the vFPGA.

- Finally, *the clocking management* could be considered as a cross-layer component or as part of the static logic layer. It is a physical component that manages several clock domains for all layers. It could be configurable, so the FPGA hypervisor can change the clock frequency of each vFPGA according to the configured CCM specifications.

To illustrate the abstraction level the network controller, we compare it with controllers in other works. Byma et al. [9] did a layer 2 encapsulation (i.e. MAC layer). Their platform fits more for packet processing systems as shown by the test cases in Byma's thesis [63] such as load balancing and extending SDN capabilities using vFPGAs. It can only provide infrastructure-as-a-Service (IaaS). Catapult [41] and Tarafdar [11] do layer 3 (i.e. network layer) by doing UDP-like encapsulation. Catapult takes care of having a lossless transmission using Ethernet flow control (802.1Qbb). In our platform, we raise the abstraction to layer 4 and 5 (i.e. transport and session layer of the conceptual OSI model) by using the TCP stream and manage the sessions inside the controllers. The network controller in our platform produces data stream plus the index of the targeted vFPGA. This raises the abstraction level and enables introducing computation as a service instead of IaaS.

The network controller can be implemented as off-chip to save the FPGA resources. It can also be implemented inside the FPGA. According to a commercial implementation [65] of 10 Gigabit Ethernet network controller on Xilinx Zynq Ultrascale+ MPSoC ZU9 FPGA, full stack (UDP and 2 TCP engines) consumes around 30k LUTs and around 10k LUTs is needed for each additional TCP engine. The Virtex 6 FPGA used in our lab contains around 340k LUTs.

The TCP transmission protocol is chosen because of its lossless transmission and packet reordering for long data streams. In addition, we would like to match the ecosystem data format and interchanging protocols by using off-the-shelf socket connection. However, in the case of changing the needs of the ecosystem, we may use different networking controller that match its networking protocols. For example, internet of things (IoT) networking uses IPv6 for interchanging data with small packet sizes. In this case, we can replace the network controller only and produce a suitable virtualization platform for IoT.

5.2 FPGA Cloud Architecture

Our platform framework for on-cloud data processing using virtual FPGAs is shown in Figure 5.2. The framework includes the following components:

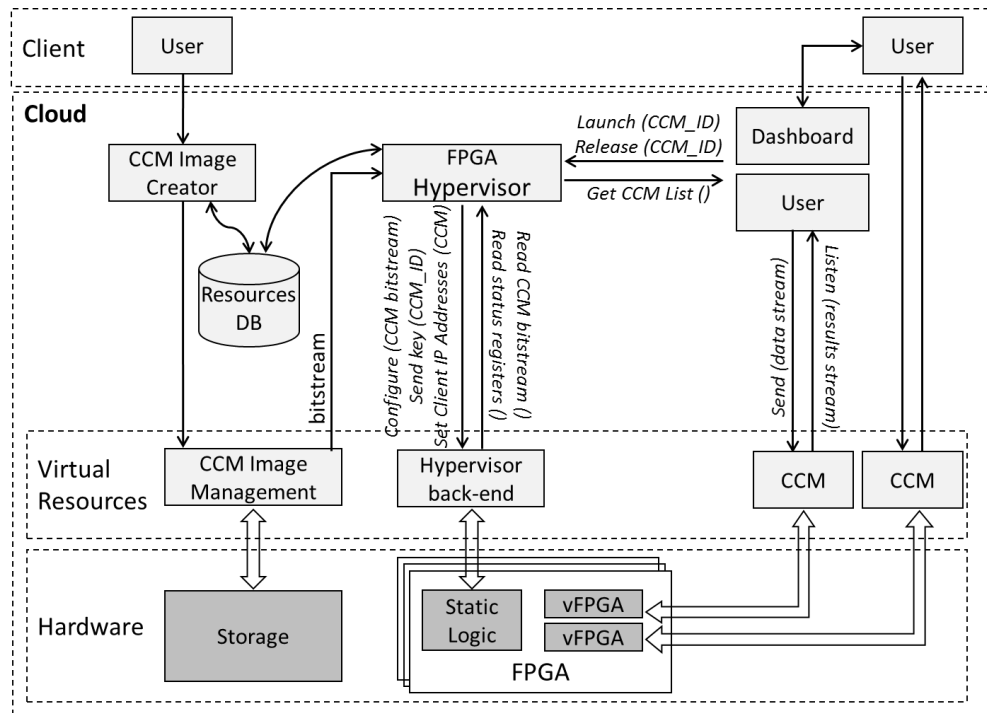


Figure 5.2: Proposed FPGA Cloud Architecture

- 1) The cloud infrastructure consists of FPGA hypervisor, resources database, CCM creator and other components. The *FPGA hypervisor* is used to manage vFPGAs resources, launching and terminating CCMs. It requests CCM bitstream from the

cloud storage management. The *resources database* stores information about CCMs and vFPGA for the management process. The *CCM creator* provides CCM creation service. It receives hardware cores written in HDL with their specification files and produces CCMs that can be implemented and sold by the cloud operator (CO) or other third parties as new services. It updates the resources database with the new CCM information.

- 2) FPGAs which are network-attached devices connected to the internal cloud network. Each FPGA contains a static logic and one or more virtual FPGAs (vFPGAs). The static logic on the FPGA is also known as FPGA hypervisor back-end. Each vFPGA can be configured with a CCM bitstream and acts as a compute node or part of a compute node on the cloud.
- 3) Software library defines the necessary API functions to manage and use virtual FPGAs. The API functions are represented by the arrows going in and out of the cloud components in Figure 5.2. The FPGA hypervisor provides API functions to launch, release CCMs. The FPGA hypervisor back-end defines API functions to be used by hypervisor front-end like configuring specific vFPGA, reading status registers, set client IP address, etc. The client can conventionally request to launch the CCM and initiate the data sending, processing and receiving. The client can use the Send/Receive, which are socket-based API functions, to access a CCM.

The CCM can be used by a client through the dashboard or by software within the cloud.

5.3 FPGA hypervisor

The FPGA hypervisor manages vFPGA resources and CCM images. It launches, terminates CCMs and keeps track of available vFPGAs and CCM with the help of the

resources DB. The static logic in each FPGA manages vFPGAs in that FPGA so we refer to it as a hypervisor back-end.

The *resources DB* is a database that stores vFPGA and CCM management information such as occupied/free vFPGA resources, user-CCM relationship, vFPGA-CCM relationship, etc. Virtual FPGA (vFPGA) is the reconfigurable region on the physical FPGA that can hold a CCM. vFPGA is configured by a CCM bitstream. CCM is the application hardware configured on a vFPGA. CCM might have several bitstreams to match each different vFPGA type.

FPGA hypervisor provides API functions for launching, using and releasing vFPGAs. Table 5-1 lists the main API functions provided by the software library of our platform. Accessing the CCM is done through only two functions; one for transmitting data and the other for receiving results.

Table 5-1: Main API function in the software library.

Function name	Communication type	Category	Implementation
Send (data stream) Listen (results stream)	Socket (TCP Stream)	User-to-CCM API functions	Put data on TCP packets
Launch (CCM_ID) Get CCM List () Release (CCM_ID)	Message Passing	User-to- Hypervisor API functions	Use the cloud message passing format that is used to launch VMs
Configure (CCM bitstream) Read CCM bitstream ()	Socket (TCP Stream)	Hypervisor-to- Hypervisor back-end API functions	Put data on TCP packets
Read status registers () Send key (CCM_ID) Set client info (Sender IP Address , Receiver IP Address) Set Parameters()	Socket (UDP)		The UDP packet contains command No. and command value.

5.3.1 User-to-CCM API functions

Accessing CCM is done through only two functions one for transmitting and the other for receiving. An implementation example of two functions using python is shown in Figure 5.3. The function “*Send (data stream)*” establishes a TCP stream session and sends the data stream over the session. The function “*Listen (results stream)*” establishes a listening TCP stream session and collects the results. The user should call the listener first then he sends his data. The CCM hardware receives a reset signal with the creation of each TCP session.


```

import socket
BUFFER_SIZE = 1024

def Send_data(SERVER_IP, SENDING_PORT, data):
    conn_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn_server.connect((SERVER_IP, SENDING_PORT))
    conn_server.send(data)
    conn_server.shutdown(socket.SHUT_RDWR)
    conn_server.close()

def Listen_to_results(SERVER_IP, RECEIVING_PORT):
    conn_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn_server.connect((SERVER_IP, RECEIVING_PORT))
    data=''
    while len(data)<100
        data = data + conn_server.recv(BUFFER_SIZE)
    conn_server.shutdown(socket.SHUT_RDWR)
    conn_server.close()
    return data

```

Figure 5.3: Python implementations for the functions “Send (data stream)” and “Listen_to_results (data stream)”. Both functions use TCP stream socket and require the CCM IP address and port number.

5.3.2 User-to-Hypervisor API functions

The hypervisor functions are message passing functions. The same functions used in other cloud platforms to request launching and releasing virtual machines are adapted for requesting launching and releasing virtual FPGAs. The function *“Launch (CCM_ID)”* sends a message to the hypervisor to launch a CCM. Once the hypervisor receives this message, it looks for the CCM_ID in the database and looks for a suitable free vFPGA. Then, it fetches the suitable bitstream image from the storage. Then it uses the function *“Configure (CCM bitstream)”* to download the CCM image. Finally, the hypervisor sends a message to the client to inform him about the IP address of the launched CCM. The function *“Release (CCM_ID)”* sends a message to the hypervisor to release the vFPGA resources of a CCM. When the function *“Get CCM List ()”* is called, the hypervisor using the resources DB builds a CCM list as long with their unique CCM_IDs and description. If the user needs secure communication, he interchanges an encryption key with the FPGA hypervisor using Diffie–Hellman key exchange method. Then, the hypervisor uses the

command “*Send key (CCM_ID)*” to send the key to the hypervisor back-end. The encryption and decryption engines use that key to decrypt incoming data and encrypt outgoing results. The hypervisor front- and back-ends take care of removing the key with each change on the IP sender and receiver.

5.3.3 Hypervisor-to-Hypervisor back-end API functions

The hypervisor back-end functions are socket functions between the hypervisor front-end and back-end. There are two functions that use TCP stream socket-based communication. The function “*Configure (CCM bitstream)*” downloads a partial bitstream that represents a CCM image on the FPGA. It uses TCP stream socket connection and works like the function “*Send (data stream)*”. The function “*Read CCM bitstream ()*” reads back the CCM bitstream which is useful for supporting CCM migration. It uses TCP stream socket connection and works like the function “*Listen (results stream)*”. The remaining functions use UDP socket connections.

The rest of the functions uses UDP socket-based communication. The hypervisor sends UDP packet and the hypervisor’s backend replies with a UDP packet. Figure 5.4 shows the UDP packet format which contains sequences of the CMD/Value pairs in its payload. The CMD/Value represents a function number and value. The function “*Read status registers ()*” reads information about the running CCM status. The function “*Send key (CCM_ID)*” changes the encryption/decryption key of the CCM. The function “*Set client info (Sender IP Address, Receiver IP Address)*” changes the sender and the receiver IP addresses of the CCM. The function “*Set Parameters ()*” is used to configure some registers with specific values. One example is the frequency register that determines the CCM operating frequency.

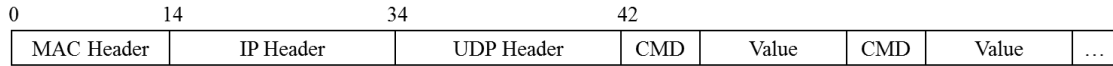


Figure 5.4: Hypervisor to hypervisor’s backend functions uses UDP socket connection. The UDP payload contains a sequence of CMD and value pairs. Several commands can be sent on one UDP packet. Hypervisor sends a UDP packet and hypervisor’s back end reply with a UDP packet.

5.4 A scenario of Launching, Using and Terminating a CCM

In the following, we explain how commands of the software library of our platform work.

We assume a user wants to launch and use a specific CCM. The user issues the four commands listed in Figure 5.5.

```

User commands:
  IP Address = Launch CCM (CCM_ID);
  results = Listen ((IP Address , Receiving PORT NO));
  Send ((IP Address, Sending PORT NO), data stream);
  Release CCM (CCM_ID);

```

Figure 5.5: Scenario of using a CCM on a cloud computing system. The user issues four commands to launch, send data, receives results, and terminate CCM.

In the following, we list all in cloud steps done to serve the user request. The message sequence diagram is shown in Figure 5.6.

1. IP Address = Launch CCM (CCM_ID)
 - 1.1. It gets the CCM information and available free vFPGAs from the “Resources DB”.
Then, it decides which vFPGA is going to be used and requests the CCM image from the “CCM image management”. We are assuming that the user already knows the CCM_ID. The user can get the CCM_ID using the command *Get CCM List ()*.
 - 1.2. The hypervisor gets the IP address and port number of the specific FPGA, and other network parameters from the Dynamic Host Configuration Protocol (DHCP) server and then executes the internal function “**Send ((IP Address, Port_no), bitstream)**”.
 - 1.3. The hypervisor back-end configures the vFPGA with the CCM image

- 1.4. The hypervisor internally issues the command “**Set Client IP Addresses (Sender IP Address, Receiver IP Address)**” to configure the sender and receiver client IP addresses in the hypervisor back-end.
- 1.5. The hypervisor back-end opens a listener to receive CCM inputs and starts another TCP session for sending the results.
- 1.6. The hypervisor returns the CCM IP address to the user. The hypervisor-back end is never revealed to the user.
2. results = Listen ((IP Address, Receiving PORT NO))
 - 2.1. The command is executed in the user machine to start the listening session. It is usually executed as a new thread, so the program can overlap sending data and receives results.
3. Send ((IP Address, Sending PORT NO), data stream)
 - 3.1. The user sends the data to the CCM. The command is executed in the user machine. It establishes a TCP stream session, sends the data to the CCM and terminates the session.
4. Release CCM (CCM_ID)
 - 4.1. The hypervisor executes an internal command “Send ((IP Address, Port_no), empty bitstream)”
 - 4.2. The hypervisor back-end configures the vFPGA with the blank CCM image
 - 4.3. The hypervisor back-end clears the registers of the sending and the receiving IP addresses.
 - 4.4. The hypervisor updates the “Resources DB” and marks the vFPGA resource free.

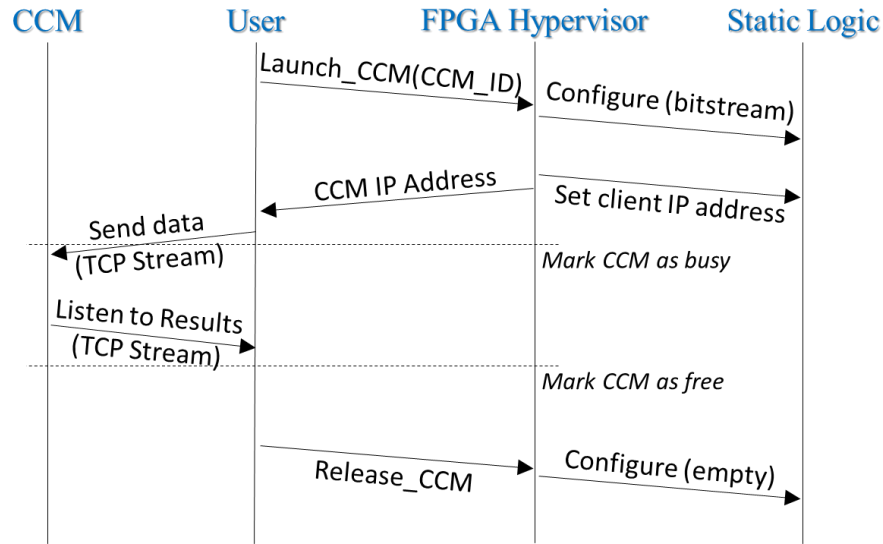


Figure 5.6: Scenario of using a CCM on a cloud computing system. First, the user requests to launch a CCM. The FPGA hypervisor configures a vFPGA with the CCM bitstream and returns the CCM IP address back to the user. The user interacts with the CCM by sending data and receiving results. Finally, the user releases the CCM.

5.5 CCM Creation

The platform introduces CCM as a service in which CCMs can be implemented and sold by the cloud operator (CO) or other third parties to the client. The CCM provides synthesizing tools for the FPGAs used in the cloud system. The CCM creation process is depicted in Figure 5.7. First, a designer uses a high-level synthesis tool (HLS) and hardware description language (HDL) to create a hardware and send it to the CCM creator. Then, the designer sends his hardware (HLS/Verilog code) with an additional XML file that describes the hardware I/Os. Another Vera file describes how data should be applied to the hardware to the CCM creator. The creator uses automatic tool that creates a wrapper for this hardware to match its interface with the vFPGA standard interface. The whole hardware is synthesized several times to generate several partial bitstreams for the hardware design each one matches a different vFPGA. After generating partial bitstreams they are stored on the cloud storage. The resource database is updated with the new CCM and its generated bitstreams information like the CCM ID, file name, and vFPGA ID.

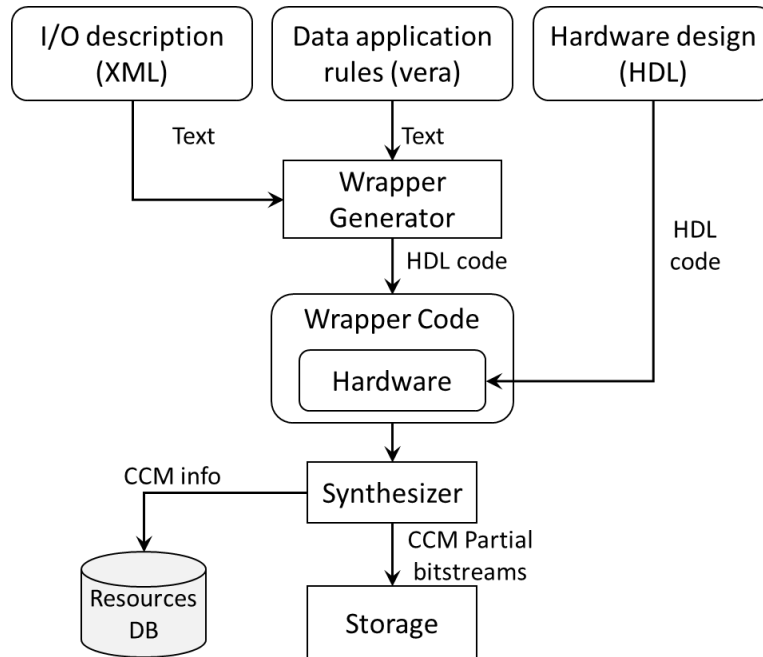


Figure 5.7: CCM creator receives hardware design (HLS/HDL), XML file describes hardware I/Os, and Vera file describes how data is applied. Then, it creates a CCM, synthesized and generates partial bitstreams. Finally, it saves the bitstreams in the cloud storage and their info in the Resources database.

5.6 Properties of the Platform

In this section, we illustrate the computation model of our platform. Then, we list cloud properties supported by our platform and explain how those properties are achieved.

5.6.1 The platform computing model

The computing model of our platform is a streaming computing model. The CCM is a network-attached computing machine that has a receiving port and a transmitting port. Sending data to the CCM and receiving results from the CCM is done through the usual networking protocol used in the cloud. In the current version we have chosen the TCP Stream as the main communication protocol with the CCM for two main reasons:

- 1- The recent software libraries that implement the TCP stream rely on the data plane development kit (DPDK) which allows fast packet processing. Using the DPDK allows the software side to send and receive data at high speed and can approach the theoretical speed of the communication link.

- 2- TCP protocol guarantees packet ordering. This makes the application feels as it received a continuous long stream. This also removes the need to build a custom reordering mechanism in the static logic on the FPGA and on the software side.

In ideal cases, the CCM consumption rate matches the data incoming rate. we do not assume a large buffer that stores unlimited data. When the CCM consumption rate is faster than the data incoming rate, the CCM is stalled waiting for data arrival. When the CCM consumption rate is slower than the data incoming rate, the CCM issues a back-pressure signal which propagates to the Ethernet controller through AXI interfaces. The platform consists of several layers with AXI interfaces between every two layers as explained in section 6.1 and illustrated in Figure 6.4. When the network controller receives the back-pressure signals, its buffer becomes full and it starts dropping packets. The TCP stream does not acknowledge the non-received packets and therefore retransmission is done.

Adjusting clock frequencies in different clock domains in the platform is important to control the produce-consume model of the system. The static logic should be clocked at least at the Ethernet controller clocking speed (i.e. 156.25MHz for 10GE or 125MHz for 1GE). Since packet headers and interpacket gaps are discarded (which represents 60 bytes at least), the static logic could work fine at a little bit lower frequency (e.g. at 150MHz for 10GE or 120MHz for 1GE).

5.6.2 Abstraction

The virtual FPGAs are standalone resources which are completely disaggregated from servers. This simplifies their management and renting them as standalone resources. The CCMs are standalone network-attached computing devices. Their interfaces are clearly defined and standardized to match the common data interchanging methods in the cloud.

The CCM user is not aware whether this machine is a hardware or a software machine since the data and results are sent over well-known network protocols. The user does not need to adapt the data for each specific CCM. No need to add timing and control signals as hardware designers do. The wrapper within the CCM is responsible for adapting the data to the hardware input.

5.6.3 Sharing

A CCM can be shared among several users by interleaving computation sessions. The computation session is an atomic operation that cannot be interrupted. When a user uses a CCM, the hypervisor prevents other users from using it. When the current user's TCP session(s) to the CCM terminate, another user can request the same CCM and the hypervisor restricts its use to the new user for one session and so on. With each session, the whole CCM is reset. The CCM's serializer and deserializer take care of flushing all results out before terminating the session.

5.6.4 User data security

If the user requires a secure channel to the CCM, (s)he exchanges a symmetrical encryption key with the FPGA hypervisor using Diffie–Hellman key exchange. Then, the hypervisor uses the function “Send key (CCM_ID)” to send the key to the hypervisor's back-end. The encryption and decryption engines use that key to decrypt incoming data and encrypt outgoing results. The hypervisor front- and back-ends take care of removing the key with each change in the sender's and receiver's IP.

5.6.5 CCM clusters on Multi-vFPGA

A cluster of network-connected CCMs can be created and saved as a new CCM. CCM network can be built by carefully setting the sending and receiving IP addresses of each CCM in the cluster. For example, an FPGA chain can be created by setting the receiving

IP address of each FPGA in the chain as a sending IP address for its previous vFPGA. The receiving address of the first vFPGA and the sending addresses of the last vFPGA in the chain becomes addresses for the resulted CCM. The new CCM information is stored in the Resources DB with pointers to the information of other CCMs constructing it.

CHAPTER 6

FPGA Virtualization Platform

Our proposed virtualization platform is based on partial dynamic reconfiguration. The physical FPGA is divided into a static region (that is kept as is with no reconfiguration), one or more dynamically reconfigurable regions, and a communication controller. Each dynamically reconfigurable region corresponds to one vFPGA where a user's design can be placed (along with the wrapper). Our specially developed wrapper controls clocking the user design according to data arrival. An overview of the proposed platform is shown in Figure 6.1. It has four distinct layers; a network layer, static logic, wrapper(s), and user design(s). The network controller handles physical connections and establishes TCP network sessions between the users and their designs in the virtual FPGAs. It manages the MACs and IP addresses assigned to the vFPGA. This enables users to use their vFPGA-based applications like any standard server; sending requests (input data) to the assigned IP (in this implementation we are ignoring ports, though later we may direct traffic to a

specific user's sub-circuit based on port number). The static logic routes TCP payloads between vFPGAs and the network controller. It also contains clock management resources that generate controllable clock domains for each vFPGA and the re-configuration management logic that can download a user's design at run time to one of the vFPGAs. The re-configuration management unit has its own MAC/IP addresses to receive the partial reconfiguration bit streams and reconfigure the vFPGA regions. Having its own MAC/IP address allows it to be integrated with the cloud/DC management tools as a 'Reconfiguration Server'. The wrapper has a fixed interface to the static logic and a custom (automatically-generated) interface to the user's design allowing it to fit into a vFPGA.

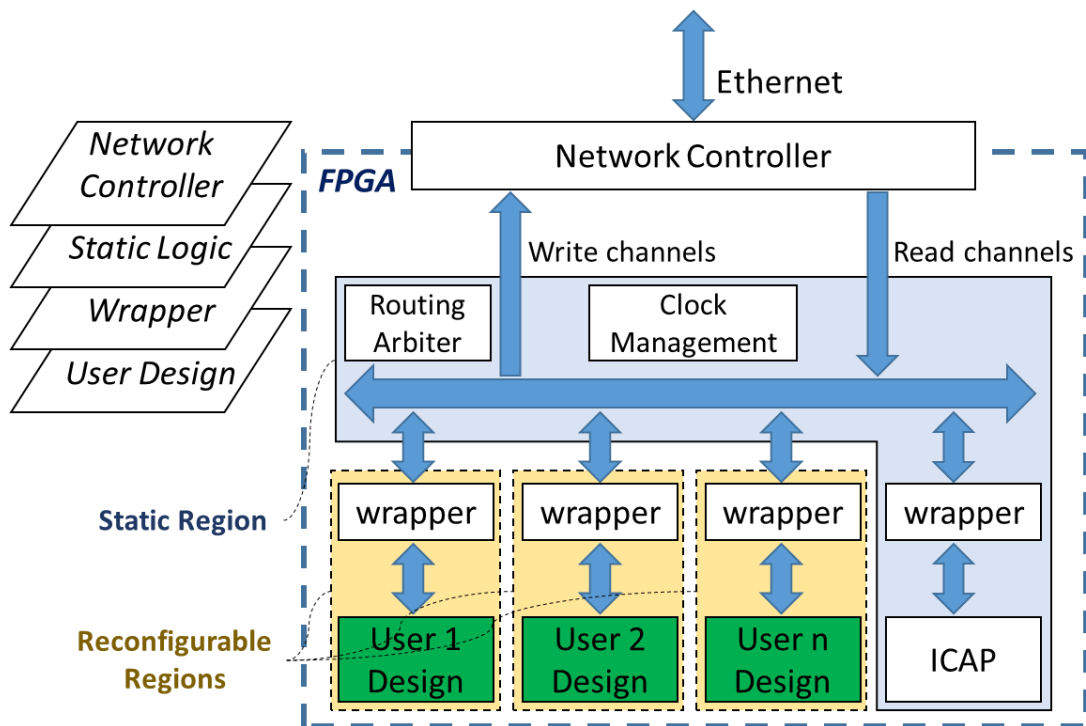


Figure 6.1: Virtualization platform overview. FPGA is partitioned into a static region and several reconfigurable regions to be used as virtual FPGAs.

6.1 Data Communications

Data movement between the first three layers of the platform follows the standard two-way handshaking mechanism as defined in AXI4 stream specifications [48]. This enables both

reader and writer to control the data transmission rate and to communicate without losing any cycles. Figure 6.2 shows the timing diagram of the AXI interface. The data transfer only when both TVALID (from the source) and TREADY (from the destination) are high.

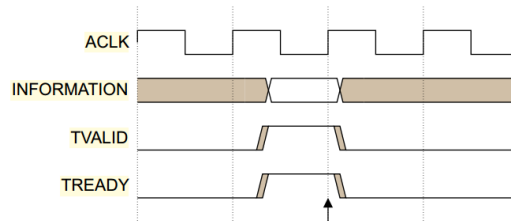


Figure 6.2: Timing diagram of the two-way handshaking process [48].

Cross clock-domains data movement is achieved with *asynchronous FIFOs*, Figure 6.3. There are two asynchronous FIFOs in the Ethernet controller and another two per each wrapper as shown in Figure 6.6. These FIFOs are implemented with embedded FPGA RAM blocks (BRAMs) with AXI read/write interfaces. The writing ports of the FIFOs are directly mapped to AXI write channels. Reading, however, is not straightforward since BRAMs require two clock cycles for the first read (then one cycle per other consecutive reads). Hence a pre-fetch circuit and a control logic were added to guarantee correct AXI timing (one read per cycle).

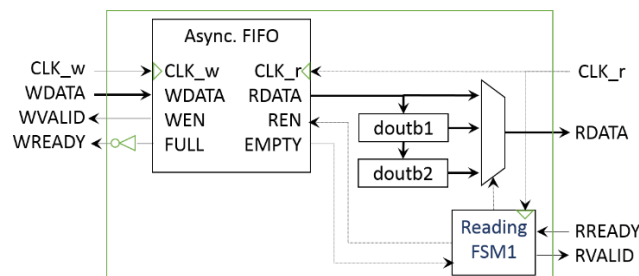


Figure 6.3: BRAM-based asynchronous FIFO for transferring data across unrelated clock domains using the AXI interface.

The interface between every two layers is clearly identified and presented in Figure 6.4. the interface between the network controller and the static logic consists of read- and write-

AXI channels associated with the virtual FPGA index. The network controller sets the rx virtual FPGA index according to the MAC address of the received packet. The router in the static logic sets the tx virtual FPGA index and forwards the vFPGA results to the network controller. The network controller sets the MAC address of the transmitted packets according to the tx vFPGA index. The interface between the static logic and each vFPGA region consists of clocking signals and read and write AXI channels. The clocking signals include the wrapper's clock, the static logic's clock, user-design's clock, and clock-enable and wrapper's reset. All data buses in the AXI read/write channels have the same width which depends on the Ethernet interface underuse. For example, 1 G Ethernet introduces 8-bit words while 10 G Ethernet introduces 64- or 128- bit words.

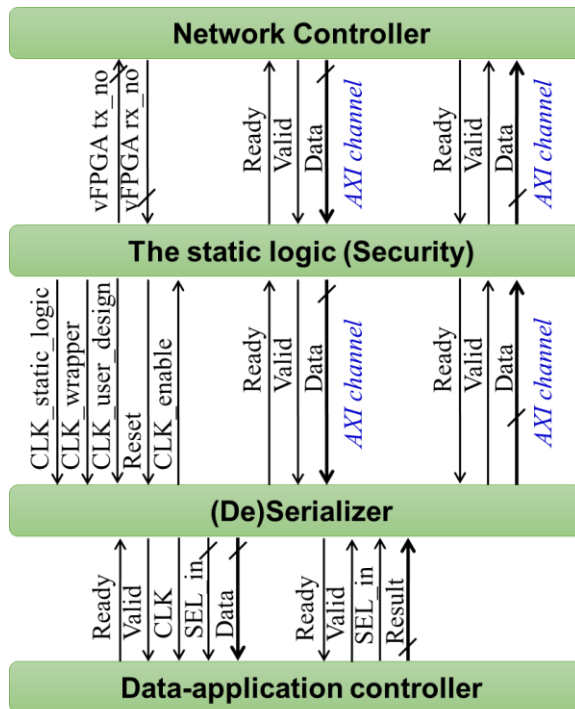


Figure 6.4: Inter-layer interfaces. The interface between L1 and L2 consists of two AXI interfaces and the virtual FPGA indices. The interface between L2 and L3 consists of two AXI interfaces and clocking signals. The interface between L3 and L4 consists of two AXI interfaces whose data have the internal wrapper formats.

The AXI interface is used everywhere in the platform. The static logic rx router is an AXI interconnect between one master (data are coming from the Ethernet controller) to several

slaves (data are routed to vFPGAs). The static logic tx router is an AXI interconnect between several masters (vFPGAs) to one slave (Ethernet controller). The wrapper components such as packing, unpacking, serializer, deserializer, and asynchronous FIFOs all of them communicate through AXI interfaces.

AXI interface can issue a back-pressure signal and this signal propagates through all AXI interfaces in the data path. This explains how our system controls the data stream flow. If the incoming data rate is greater than what the application can consume, the busy application' back-pressure signal is propagated to the network controller. The network controller starts dropping packets and the TCP protocol retransmits them. This way the back-pressure propagates to the user. It is the user responsibility to send data according to the consuming rate of the CCM. The CCM designer should provide information about the CCM throughput.

6.2 Network Controller

The network controller implements the TCP's data link, network, and session layers of the OSI network stack. It establishes sessions between vFPGAs and their users and ensures data ordering and correctness. It receives configuration bitstreams and users' data, deliver it to the static logic and transmit the results back to the user. More precisely, it performs the following tasks:

- Establishes and terminates TCP sessions between vFPGAs and their users. It stores source addresses and other session data.
- Forwards the payload of the received TCP packets to the static logic associated with the target vFPGA index.

- Constructs TCP packets for the received results from vFPGAs and transmits them to their users.
- Stores and manages MAC/IP addresses for all associated vFPGAs and negotiates for dynamic network addressing using the DHCP protocol.
- Announces the existence of associated vFPGAs over the network and replies to network queries about vFPGAs such as ARP and ping requests.

The network controller can either be integrated with the static logic in the physical FPGA or it can be an off-the-shelf device external to the FPGA. It is also possible to share a network controller among several vFPGAs using a single Ethernet cable connected to the physical FPGA or associate one network controller per vFPGA such that each vFPGA will have a dedicated Ethernet cable connected to the physical FPGA.

The fixed interface between the static logic and the network controller consists of AXI read-data channel, write-data channel, and the vFPGA indices. The data width is 8/64 bits for the 1 GE/10 GE Ethernet interface, respectively. Asynchronous FIFOs are used to move the data across the three clock domains of the static logic, the Ethernet transmitter, and the Ethernet receiver.

Figure 6.5 shows our Ethernet controller designed to achieve the maximum throughput. The receiver works as follows; the Phy_rx receives Ethernet packets and check the packet's CRCs. The sniffer reads packet header on-the-fly, check addresses, and trigger the suitable reaction. If a TCP packet is received, its payload is stored in the rx-Async buffer. The asynchronous buffer depth can hold two packet-payloads. So, a packet can be read while the other packet is buffered. The transmitter works as follows; once the tx-Async buffer

has a ready payload, it triggers the finite state machine controller to start constructing an Ethernet packet and choose the suitable Ethernet header. Phy_tx transmits the packet and adds a preamble, CFD, and CRC to it.

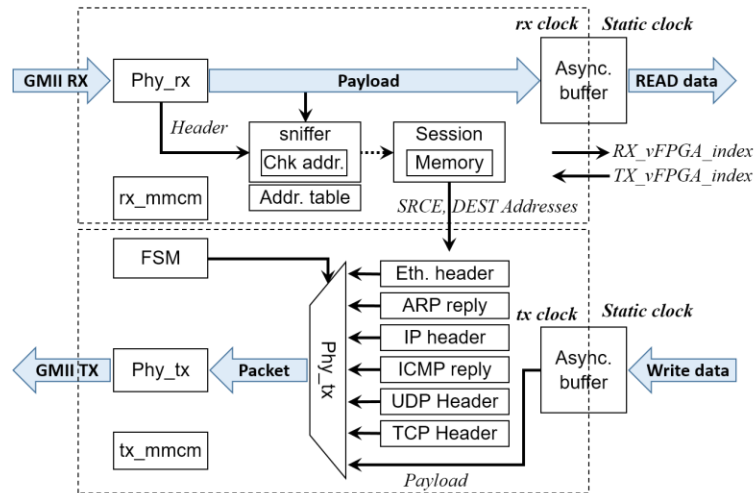


Figure 6.5: The implementation of the Ethernet controller.

6.3 Static logic

The static logic includes data routers, a reconfiguration management unit, a data security unit (optional), and a clock management unit. These components are described next.

6.3.1 Data routing

Data routing is needed when the network controller is shared among several virtual FPGAs. Routing data between the network controller and vFPGAs is done through two AXI interconnects. The first one reads from the rx-Async buffer and route to the corresponding vFPGA. The second AXI interconnect reads results from one vFPGAs at a time and forward them to the tx-Async buffer. The result of each vFPGA is collected separately to guarantee no interference with other vFPGAs outputs.

6.3.2 Reconfiguration management unit (RM)

Reconfiguration manager (RM) receives partial bitstreams over the Ethernet to reconfigure any of the vFPGAs. It has its own MAC/IP addresses and the network controller deals with it as another vFPGA. It consists mainly of an internal reconfiguration access port (ICAP) surrounded with a wrapper. It also responsible for freezing the partial region I/O interfaces during the configuration.

6.3.3 Clock management unit (CM)

The clock management unit produces several clocks for the different domains as shown in Figure 6.6 The Ethernet controller has two separate clock domains; one for the transmitter and another for the receiver. The static logic has its own clock domain. Each user design is clocked by a dedicated controllable clock signal. Though the wrapper that surrounds the user design uses the same frequency, it has a separate clock domain than the design. Finally, part of the wrapper shares its clock with the static logic. These separate clock domains allow optimum operation of different parts of the system independently from other parts, users to set up the frequency of their circuits in the vFPGA, and most importantly, the static logic does not need to be re-synthesized with the users' design every time a new user's design is loaded into a vFPGA. The last point is essential for virtualizing the FPGA among multiple users.

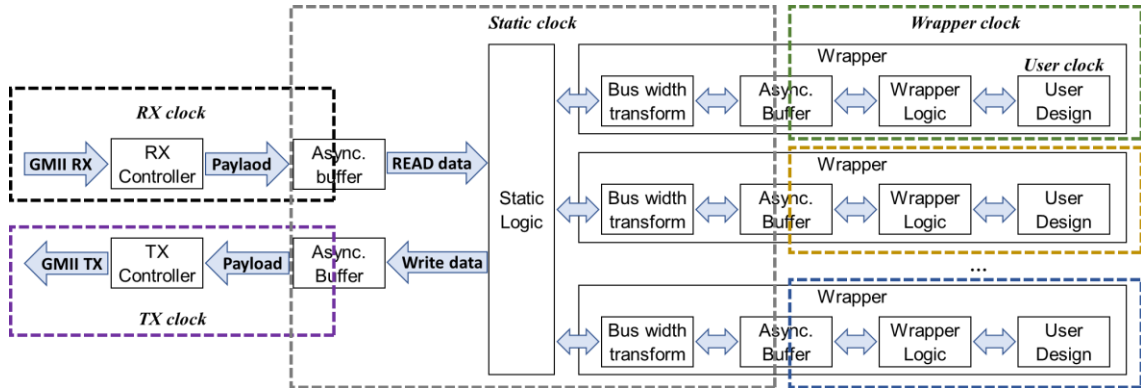


Figure 6.6: The platform's different clock domains and the use of asynchronous buffers to move data across these domains.

Standard clock buffers and clock management units (CMUs) available on commercial FPGAs have many properties that are utilized in the wrapper design. First, they are controllable (i.e. stoppable). The wrapper uses this property to stall and release the user design clock according to the availability of input data and other conditions. Second, they are run-time reconfigurable, allowing the wrapper to set the user design's clock frequency at run-time. Third, their clock phases can be shifted by 180 to provide negative edge clocking for the user design.

The static logic should be clocked at least at the Ethernet controller clocking speed (i.e. 156.25MHz for 10GE or 125MHz for 1GE). Since packet headers and inter-packet gaps are discarded (which represents 60 bytes at least), the static logic could work fine at a little bit lower frequency (e.g. at 150MHz for 10GE or 120MHz for 1GE).

6.4 The wrapper's design

Ideally, users would want to configure their application circuit on a data center-attached vFPGA, and then use it by sending input data streams and receive output data streams through Ethernet packets. The wrapper allows users to fit, communicate, and control their designs in any partially reconfigurable region (i.e. vFPGA). It is automatically generated for each design according to a user-provided XML input/ output specification. The designer

also prepares a description of the data format and application/capture rules using a subset of the verification language OpenVera (SystemVerilog) [66]. The wrapper generated from the user's specifications (XML and Vera) provides the interface between the user's design from one side and the static logic from the other side. The user-specified control is incorporated into the wrapper's design itself. This means users can pack and send/receive their raw data/results to the FPGA as dense packets with no embedded control data to utilize the maximum communication bandwidth.

A custom wrapper is generated for the user's design based on a user-provided XML specification which is then synthesized with the design to produce the partial configuration bitstream of the design. Several bitstreams could be generated for each of the different vFPGA instances available on all the FPGA types attached to the cloud (hence, a user's design can be seamlessly migrated between any vFPGA on the cloud).

In the XML description, users can divide their circuits' inputs and outputs into groups such that one input/output group is applied/captured at each clock cycle. If there is more than one output group, the design is stalled until all groups are captured.

In the Vera description, users specify how input/output groups are applied and captured. So, a user could specify one input group to be applied, then clock the circuit for a certain amount of cycles, then capture a certain output group when a certain output is changed (e.g. a Done flag), and so on. This allows for any computation semantics to be implemented. An input/output group's size could be anything from 1 to n-bits.

6.4.1 Conceptual design of the wrapper

Figure 6.10 and Figure 6.7 show the conceptual design of the wrapper and a flow chart describing its operation, respectively. When a network packet arrives, its payload is extracted by the fixed logic and sent to the vFPGA as a sequence of c -bit words ($c=8$ or 64 for 1 Gb Ethernet or 10 Gb Ethernet, respectively). Packing/unpacking circuits convert the received/ sent c -bit words to w_{in} -bit words. w_{in} represents the circuit's input data size (with no control signals) and is inferred from the user's specifications (hence, $w_{in} \geq n$). The packing/unpacking circuitry is designed to achieve the maximum throughput. If $c > w_{in}$, one output is produced per cycle, and if $c < w_{in}$, one output is produced each w_{in}/c cycles. If $c = w_{in}$, then the packing/unpacking circuits are removed from the wrapper. A serializer then groups the input data and the control signals that it generates into user-specified input groups and applies them to the circuit in the user-specified order. Input groups with sizes less than n are simply connected to the lowest bits of the serializer. The wrapper receives consecutive words with 64-bit length, combine them together, generate w_{in} -bit data words, and pass them to the serializer. The serializer generates n -bit word starts with the application bit, followed by input group index, and finally the data. The serializer output is stored in the input FIFO.

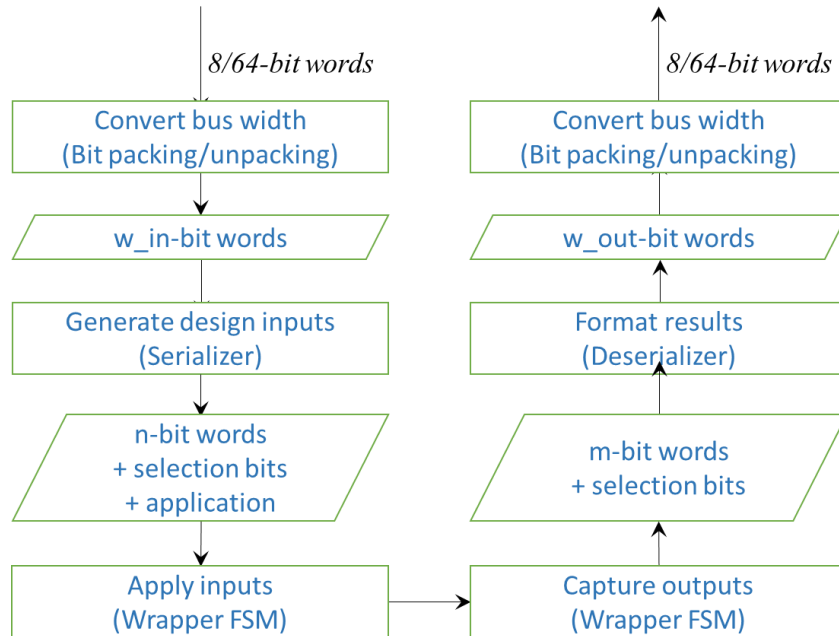


Figure 6.7: A flow chart illustrating the data flow from/to the design through the wrapper. The left-hand side shows the data input flow starting from receiving a payload of a user's network packet till its application to the design. The right-hand side shows formatting and sending the results starting from capturing the outputs till generating the payload for the network packet to be sent back to the user.

6.4.2 Wrapper components

- *Bit unpacking*: it receives w -bit words, combine them together and then produces an n -bit word, where $n > w$. If the input is streamed, it will produce a word exactly each n/w cycles. It contains slicer, state machine, and several registers as depicted in Figure 6.8.
- *Bit packing*: it receives the m -bit word, divides it to m/w words and produces w -bit word each cycle. It contains slicer, state machine, and several registers as depicted in Figure 6.8.

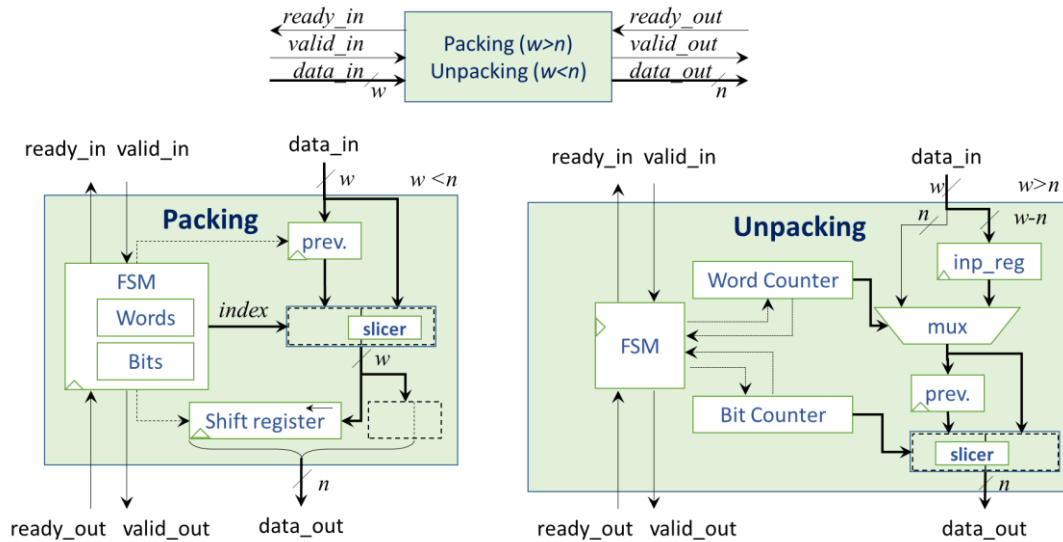


Figure 6.8: Our implementation of the Packing/unpacking circuitry. If the input data width is greater than the output data width, the packing circuitry is used. If it is less, the unpacking circuitry is used. If they are equal, the packing/unpacking part is removed.

Our implementation of the bit-packing and -unpacking circuits are depicted in Figure 6.8. Each one has a slicer, state machine, word and bit counters, and several registers as. The bit counter gives the number of shifted bits in the slicer. The word counter in the unpacking circuit determines the index of selected word from the wide input. The word counter in the packing circuit determines when a complete output becomes valid at the circuit output.

Slicer: The slicer is the main building block in the packing and the unpacking circuits in our implementation. It is a combinational circuit that receives n -bit input and produces $(n/2)$ -bit word as output. The output is just a slice of the input chosen according to the selection input. Its diagram is shown in Figure 6.9 below.

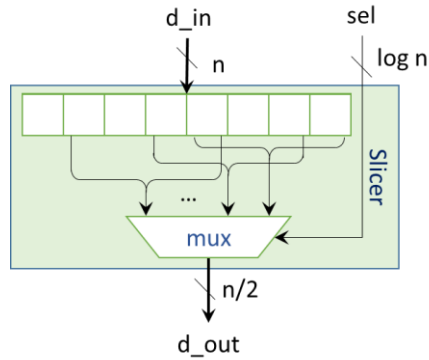


Figure 6.9: Slicer is a combinational circuit that selects $n/2$ consecutive bits from an n -bit input.

- *The serializer* translates the data into the wrapper internal data input format. The internal data format consists of three fields; clocking information, input register index, and input register value. The serializer design with an example is illustrated below.
- *The deserializer* translates the internal data output format to data. The internal data output format consists of two fields; output group index and value. If there is only one output, then there is no index. The deserializer decides what results should be transferred to the user and what is the format of these results. The deserializer circuit design is much simpler than the serializer. In several cases, its job is to forward the circuit outputs without control signals and the selection bits to the output FIFO.
- *Input registers:* The hardware inputs are divided into groups and need to be stored into input registers. This wrapper FSM guarantees that the hardware will never be clocked with wrong inputs.
- *Cycle counter* stores the number of clock cycles that should be applied with each application command. The counter is updated at run-time.
- *Mask register* is bits vector in which each bit corresponds to an output group. It indicates whether the output group should be sent to the output FIFO or should be bypassed.

- *Output arbiter* decides which output group will be captured each cycle. The arbiter is reset when a new user input arrives, then it advances once a new output is captured.

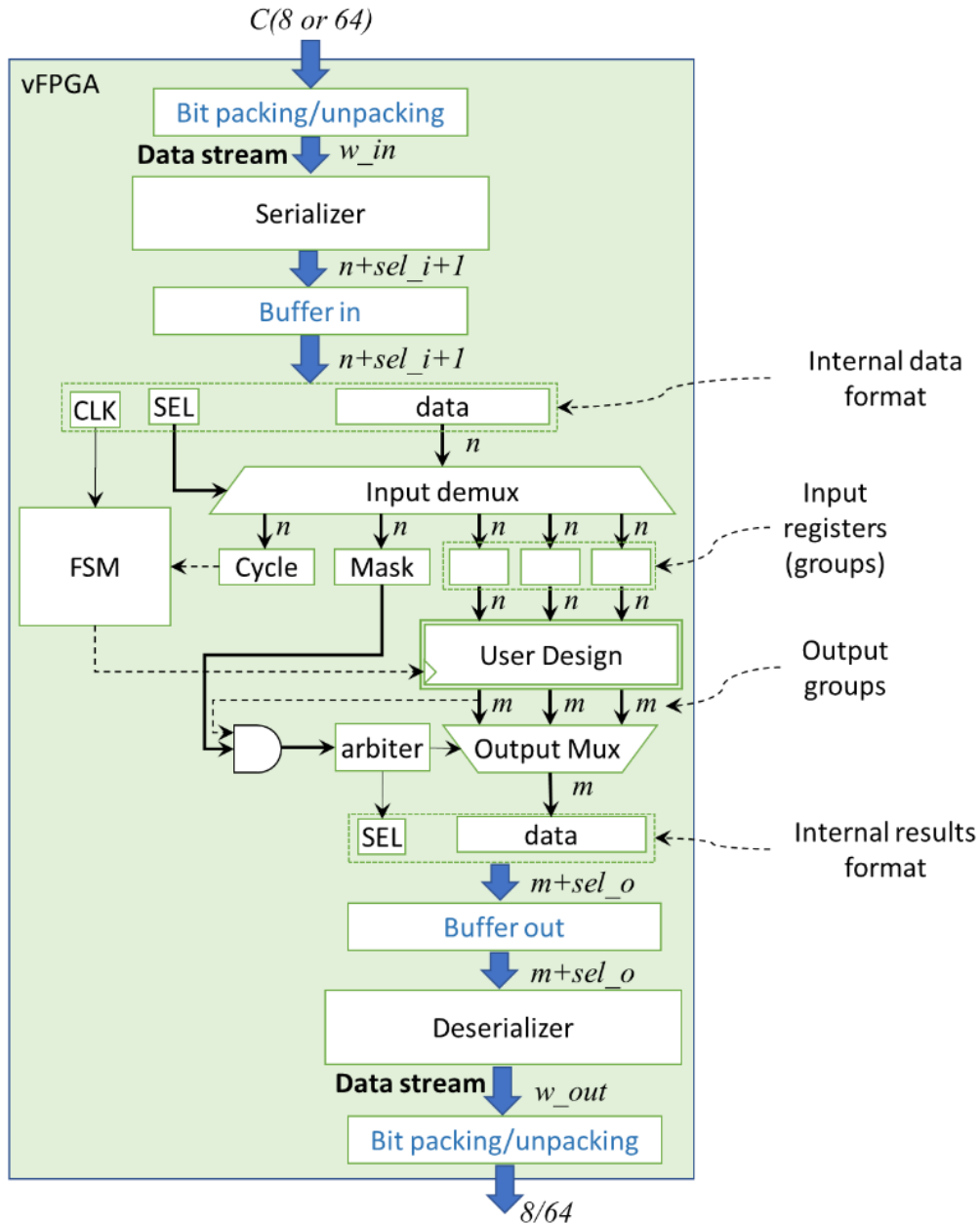


Figure 6.10: The wrapper's conceptual design.

- *Wrapper FSM* controls the clock buffer chip enable signal (CE) to control clocking the user design according to data arrival. It also controls the incoming/outcoming data to the user design by controlling ready-in signal and valid-out signal. An FSM

controls clocking the user's design according to data arrivals and user specifications, reading data from the input FIFO, applying it to the inputs, reading the output results, and capturing outputs and storing them in the output FIFO. When a new input arrives, if its clock-control bit is on, the FSM clocks the user design for the number of cycles indicated in the cycles register. The Verilog code of a wrapper FSM is shown in Figure 6.12. Figure 6.11 shows a diagram that illustrates the wrapper FSM complexity. The wrapper reaches the "counting" state if the user has specified several clock cycles per input application. At this state, the wrapper stalls inputs and stop capturing outputs until the clock count reach zero. The wrapper goes to the "One output" state if there is only one or no output to capture. This state allows receiving inputs while capturing the output. In the "Multi-output" state, the wrapper keeps capturing output without accepting new inputs until it receives the "last output" signal from the output arbiter (reading new inputs can overlap with the capturing the last output).

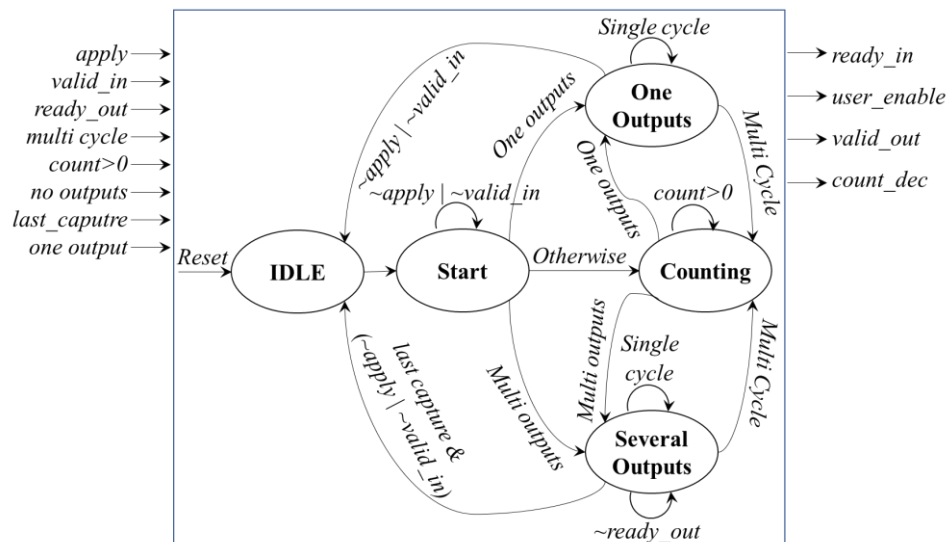


Figure 6.11: Diagram shows the complexity of building the wrapper state machine. If the hardware has one output group, then an input can be applied while capturing the outputs. If the hardware has several outputs, then the controller should flush out outputs before accepting new inputs. For some inputs, it is required to apply several consecutive clock cycles without capturing outputs or applying new inputs.

```

//apply_bit: the apply bit in the input FIFO read channel
//TVALID_in: The valid signal of the input FIFO read channel
//TREADY_out: The ready signal of the output FIFO write channel
//ZC: cycle register is zero
//Zero: cycle counter is zero
//ZO: No output is required to be transmitted to the user
//OO: Only one output is required to be transmitted to the user
//Last: Current output is the last output to be transmitted to the user

module wrapper_Moore_FSM (input clk, reset, apply_bit, TVALID_in,
                          TREADY_out, zc, zero, zo, last, oo,
                          Output TREADY_in, USER_EN, TVALID_out,
                          cntr_dec, cntr_load);

    reg [6:0] state = 7'b0;
    assign USER_EN    = cntr_dec | (state[1] & TVALID_in & TREADY_in &
                                     apply_bit & (oo | zo | (last & TREADY_out)));
    assign TREADY_in  = state[0] & ~(~zo & (~TREADY_out | ~last));
    assign TVALID_out = ~zo & USER_EN;
    assign cntr_dec   = state[3] & last & TREADY_out;
    assign cntr_load  = ~state[3];

    always @(posedge clk)
        if (reset)
            state <= 'h00;
        else case (state)
            'h00: state <= 'h03;
            'h03: if (TVALID_in & apply_bit) begin
                    if (~zc) state <= 'h0a;
                    else if (oo) state <= 'h07;
                    else state <= 'h17;
                end
            'h0a: if (zero) begin
                    if (oo) state <= 'h07;
                    else state <= 'h17;
                end
            'h07: begin
                    if (TVALID_in & apply_bit) begin
                        if (~zc) state <= 'h0a;
                    end else state <= 'h00;
                end
            'h17: if (TVALID_in & apply_bit & TREADY_out) begin
                    if (~zc) state <= 'h0a;
                    else state <= 'h17;
                end else if (last) state <= 'h00;
            default: state <= 'h00;
        endcase
    endmodule

```

Figure 6.12: Verilog code of a finite state machine of a wrapper.

The user_enable signal is used to gate the application hardware clock. FPGAs are provided with controllable clock buffers [67, 68]. Figure 6.13 shows a timing diagram that explains

the effect of the user_enable signal on the output clock. Clock resources are part of the static logic. They are not reconfigurable parts. For this reason, the wrapper interface includes the user_enable output and the user_clock input.

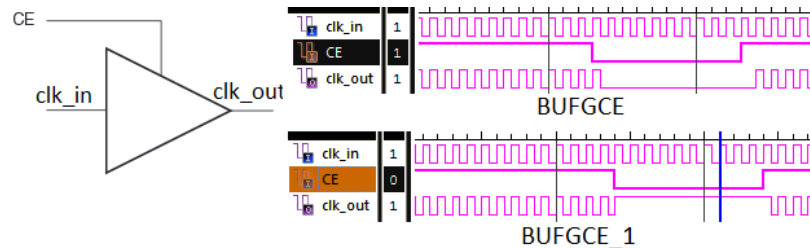


Figure 6.13: The controllable clock buffer allows controlling the application clock. When it is enabled the application runs. When it is disabled the application freezes. The upper timing diagram shows a clock buffer which always produces a low output when its enable signal is off. The lower timing diagram shows a clock buffer which always produces a high output when its enable signal is off.

By controlling this clock buffer, we can freeze the user application when the input buffer is empty and when the output buffer is full. This isolates the user application completely and gives it the feeling of having a continuous input/output stream. If the design has input groups, the controllable buffer stalls the application until setting up all input groups on each cycle. The clock cycle counter allows having several clocks for the same inputs. It also allows flushing out the results when all computations are done. The serializer decides when to change the clock cycle counter.

6.5 Wrapper generation

A wrapper generation tool is a template-based tool that has templates for all wrapper components. Some of the wrapper components are parametrizable (no need to modify its Verilog code) such as packing/unpacking circuitries, asynchronous buffers and the finite state machine (wrapper FSM). Other components are modifiable and need to be rewritten according to the XML/JSON specification file of the application hardware such as the input registers, input demultiplexer, output multiplexer, and output arbiter.

The wrapper generation process is illustrated in the flowchart in Figure 6.14. It starts by parsing the XML file to determine all parameter values and to generate the Verilog code of the modifiable components. Then, the serializer is generated according to the Vera description file. The output of the generator is a Verilog file for the wrapper which includes its modifiable components, instantiations of the parameterized components, instantiations of the serializer and the deserializer and instantiation of the user application. The wrapper file with the related Verilog files are sent to the synthesizer to generate partial bitstreams and the CCM is created.

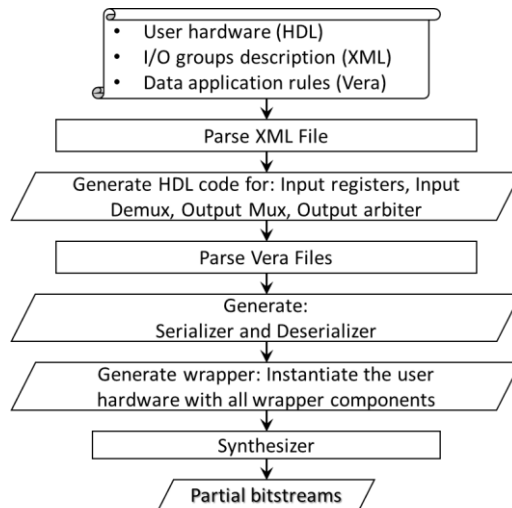


Figure 6.14: CCM creation flowchart.

Figure 6.15 shows the template-based wrapper generation algorithm. It receives the I/O data bus widths of the static logic / vFPGA interface, the specification files(XML and Vera) and design name. Lines 2 calls the parse XML algorithm which prepares the Verilog code of the modifiable components for the given design name from the given XML file. Line 3 calls the serializer/deserializer generation algorithm which returns the Verilog code of the two modules. Lines 4-13 decides whether to instantiate or not packing or unpacking circuit at the input and the output interfaces. Line 14 adds the serializer and the deserializer

modules Verilog code. Line 15 copies the Verilog file header. Lines 16-21 adds instantiations of the parameterizable components. Line 22 adds the unmodifiable components Verilog code. Line 23 adds the end module code.

Algorithm 1 Generate wrapper

```

procedure GENERATEWRAPPER(static_DATAWIDTH_IN, static_DATAWIDTH_OUT,
XMLfile, design_name, Verafile)
2:  VCode1 = ParseXML(XMLfile, design_name)
   VCode2 = GenerateTheSerializers(Verafile)
4:  if static_DATAWIDTH_IN < w_in + InputSelectionBits + 1 then
   VerilogCode = VerilogCode + Instantiate_packing_inputs()
6:  else if static_DATAWIDTH_IN > w_in + InputSelectionBits + 1 then
   VerilogCode = VerilogCode + Instantiate_unpacking_inputs()
8:  end if
   if static_DATAWIDTH_OUT > w_out + OutputSelectionBits then
10:  VerilogCode = VerilogCode + Instantiate_packing_outputs()
   else if static_DATAWIDTH_OUT < w_out + OutputSelectionBits then
12:  VerilogCode = VerilogCode + Instantiate_unpacking_outputs()
   end if
14:  VerilogCode = VCode2
   VerilogCode = VerilogCode + Wrapper_header()
16:  VerilogCode = VerilogCode + Instantiate_deserializer()
   VerilogCode = VerilogCode + Instantiate_ASYNC_FIFO_in()
18:  VerilogCode = VerilogCode + Instantiate_ASYNC_FIFO_out()
   VerilogCode = VerilogCode + Instantiate_wrapper_FSM()
20:  VerilogCode = VerilogCode + Create_clock_cycles_counter()
   VerilogCode = VerilogCode + Create_mask_register()
22:  VerilogCode = VerilogCode + VCode1
   VerilogCode = Wrapper_footer()
24:  Return(VerilogCode)
end procedure

```

Figure 6.15: Algorithm for generating the wrapper from the XML specification file and the Vera description file.

6.5.1 Parsing the XML/JSON specification file

The designer should prepare a description of his hardware inputs and outputs. In the description, input and output groups should be defined. The description should be written in XML or JSON format. These formats are chosen because they are a text-based format and they allow expressing a hierarchy in the description. The designer should prepare an XML/JSON specification file according to the schemas shown in Figure 6.17 and Figure 6.18. Table 6-1 contains the description of the XML tags and their attributes. Figure 6.16 shows an algorithm for parsing the XML specification file to generate the wrapper modifiable components.

Algorithm 2 Parse the XML description

```
1: procedure PARSEXML(XMLfile, design_name)
2:   VerilogCode = "", instantiation = "", w_in = 0, w_out = 0
3:   design_name = getDesign(XMLfile, name = design_name)
4:   for each parameter p in design_name do
5:     VerilogCode = VerilogCode + parameter_definition(p)
6:   end for
7:   InputSelectionBits =  $\log_2(\text{countInputGroups}(\text{design\_name}))$ 
8:   for each input group g in design_name do groupWidth = 0
9:     for each bus in g do
10:      if isNumeric(bus.width) then
11:        wireWidth = bus.width
12:      else
13:        wireWidth = getParameter(bus.width).value
14:      end if
15:      if (bus.start  $\neq$  -1  $\wedge$  bus.end  $\neq$  -1) then
16:        wireWidth =  $\|bus.end - bus.start\|$ 
17:      end if
18:      groupWidth = groupWidth + wireWidth
19:      VerilogCode = VerilogCode + bus.reg_declarations()
20:      instantiation = instantiation + bus.instantiation()
21:      VerilogCode = VerilogCode + BusDeclaration(i)
22:    end for
23:    w_in = Max(w_in, groupWidth)
24:    VerilogCode = VerilogCode + NewInpReg(groupWidth, load = (sel_in == g.index))
25:  end for
26:  L = countOutputGroups(design_name)
27:  if L > 1 then
28:    VerilogCode = VerilogCode + instantiate_output_arbiter()
29:  end if
30:  OutputSelectionBits =  $\log_2(L)$ 
31:  for each output group g in design_name do groupWidth = 0 mask_pin = ""
32:    for each bus in g do
33:      if isNumeric(bus.width) then
34:        wireWidth = bus.width
35:      else
36:        wireWidth = getParameter(bus.width).value
37:      end if
38:      if (bus.start  $\neq$  -1  $\wedge$  bus.end  $\neq$  -1) then
39:        wireWidth =  $\|bus.end - bus.start\|$ 
40:      end if
41:      groupWidth = groupWidth + wireWidth
42:      if (bus.mask == true) then
43:        AddToMask(mask_pin, bus.name)
44:      end if
45:      VerilogCode = VerilogCode + bus.wire_declarations()
46:      instantiation = instantiation + bus.instantiation()
47:      VerilogCode = VerilogCode + BusDeclaration(i)
48:    end for
49:    w_out = Max(w_out, groupWidth)
50:    VerilogCode = VerilogCode + AddMaskWire(g.index, mask_pin)
51:    VerilogCode = VerilogCode + AddToOutputMultiplexer(g.index, bus.name)
52:  end for
53:  VerilogCode = VerilogCode + instantiate_user_design(design_name, instantiation)
54:  ReturnVerilogCode, InputSelectionBits, OutputSelectionBits, w_in, w_out
55: end procedure
```

Figure 6.16: Algorithm for parsing the XML specification file and generating Verilog code for the modifiable parts of the wrapper.

Table 6-1: A description of the XML tags and their attributes used to describe the user hardware I/Os and their groups.

XML tag	Attributes	Description
<User_Design_List>		Contains a list of several user designs
<User Design>		Contains a list of the user design parameters, input groups, and output groups
	wrapper_name	The name of the Verilog module of the wrapper to be generated
	design_name	The name of the Verilog module of the user design to be instantiated in the wrapper Verilog code.
<Parameter>		List parameters defined by the Verilog module which is used to describe I/O bus width
	name	The name of the parameter name
	value	The value of the parameter
<Input_Group>		Define an input group which contains a list of input buses
<Output_Group>		Define an output group which contains a list of output buses
<Bus>		Define a wire, bus or part of a bus. To define a part of a bus, the start and the end indices should be used.
	name	Wire/bus name
	width	Total wire/bus width
	start	The starting index within the bus
	end	The ending index within the bus
	mask	Indicates that this bus represents a valid-out wire and it should be used to mask this group. It is used with output groups only. If the mask="true", the bus width should be 1

```

1 { "$schema": "http://json-schema.org/draft-04/schema#",
2   "type": "object",
3   "properties": {
4     "wrapper_name": {"type": "string"},
5     "design_name": {"type": "string"},
6     "Parameter": {"type": "array",
7       "items": [{
8         "type": "object",
9         "properties": {"name": {"type": "string"},
10          "value": {"type": "number"}},
11        "required": ["name", "value"]
12      }]
13   },
14   "Input_Group": {
15     "type": "array",
16     "items": [{
17       "type": "object",
18       "properties": {
19         "Bus": {
20           "type": "array",
21           "items": [{
22             "type": "object",
23             "properties": {"name": {"type": "string"},
24              "width": {"type": "string"},
25              "start": {"type": "number"},
26              "end": {"type": "number"}},
27            "required": ["name", "width"]
28          }]
29        }
30      },
31      "required": ["Bus"]
32    }]
33  },
34
35  "Output_Group": {
36    "type": "array",
37    "items": [{
38      "type": "object",
39      "properties": {
40        "Bus": {
41          "type": "array",
42          "items": [{
43            "type": "object",
44            "properties": {"name": {"type": "string"},
45              "width": {"type": "string"},
46              "start": {"type": "number"},
47              "end": {"type": "number"},
48              "mask": {"type": "boolean"}},
49            "required": ["name", "width"]
50          }]
51        }
52      },
53      "required": ["Bus"]
54    }]
55  }
56 },
57 "required": ["wrapper_name", "design_name", "Input_Group", "Output_Group"]
58 }

```

Figure 6.17: The JSON schema file for describing hardware I/Os and their groups to the wrapper generator.


```

1 <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
2           xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="User_Design">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="Parameter" maxOccurs="unbounded" minOccurs="0">
7           <xs:complexType>
8             <xs:simpleContent>
9               <xs:extension base="xs:string">
10                <xs:attribute type="xs:string" name="name" use="optional"/>
11                <xs:attribute type="xs:integer" name="value" use="optional"/>
12              </xs:extension>
13            </xs:simpleContent>
14          </xs:complexType>
15        </xs:element>
16        <xs:element name="Input_Group" maxOccurs="unbounded" minOccurs="0">
17          <xs:complexType>
18            <xs:sequence>
19              <xs:element name="Bus" maxOccurs="unbounded" minOccurs="0">
20                <xs:complexType>
21                  <xs:simpleContent>
22                    <xs:extension base="xs:string">
23                      <xs:attribute type="xs:string" name="name" use="optional"/>
24                      <xs:attribute type="xs:string" name="width" use="optional"/>
25                      <xs:attribute type="xs:integer" name="start" use="optional"/>
26                      <xs:attribute type="xs:integer" name="end" use="optional"/>
27                    </xs:extension>
28                  </xs:simpleContent>
29                </xs:complexType>
30              </xs:element>
31            </xs:sequence>
32          </xs:complexType>
33        </xs:element>
34        <xs:element name="Output_Group" maxOccurs="unbounded" minOccurs="0">
35          <xs:complexType>
36            <xs:sequence>
37              <xs:element name="Bus" maxOccurs="unbounded" minOccurs="0">
38                <xs:complexType>
39                  <xs:simpleContent>
40                    <xs:extension base="xs:string">
41                      <xs:attribute type="xs:string" name="name" use="optional"/>
42                      <xs:attribute type="xs:string" name="width" use="optional"/>
43                      <xs:attribute type="xs:boolean" name="mask" use="optional"/>
44                      <xs:attribute type="xs:integer" name="start" use="optional"/>
45                      <xs:attribute type="xs:integer" name="end" use="optional"/>
46                    </xs:extension>
47                  </xs:simpleContent>
48                </xs:complexType>
49              </xs:element>
50            </xs:sequence>
51          </xs:complexType>
52        </xs:element>
53      </xs:sequence>
54      <xs:attribute type="xs:string" name="wrapper_name"/>
55      <xs:attribute type="xs:string" name="design_name"/>
56    </xs:complexType>
57  </xs:element>
58 </xs:schema>

```

Figure 6.18: The XML schema file for describing hardware I/Os and their groups to the wrapper generator.

6.5.2 Parsing the Vera specification file

The wrapper also contains an instantiation for the serializer which should be generated by another tool as explained below. In this subsection, we just explain how the serializer and the deserializer can be automatically generated. Currently, we did not build the serializer generation tool and we did not write the algorithm of creating the serializer. Instead, we used Microsoft Excel to generate the serializer output in our experiments so, the serializer functionality is done on the user side.

The serializer generator tool translates the Vera description file to micro-instructions and stores them in the control store of the microcode, Figure 6.19. A serializer generation tool is also a template-based tool. It has a microcode template which consists of; control store, address register, output multiplexer, and loop counters. The number of loop counters varies according to the existence of nested loops in the Vera description file. We need a loop counter for each level in a nested loop. A counter can be reused in separate loops.

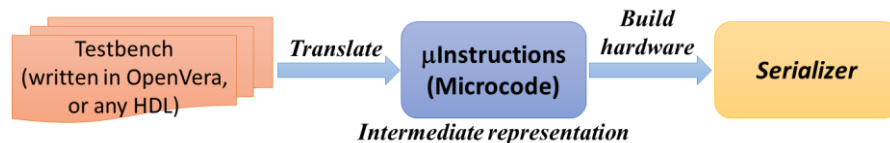


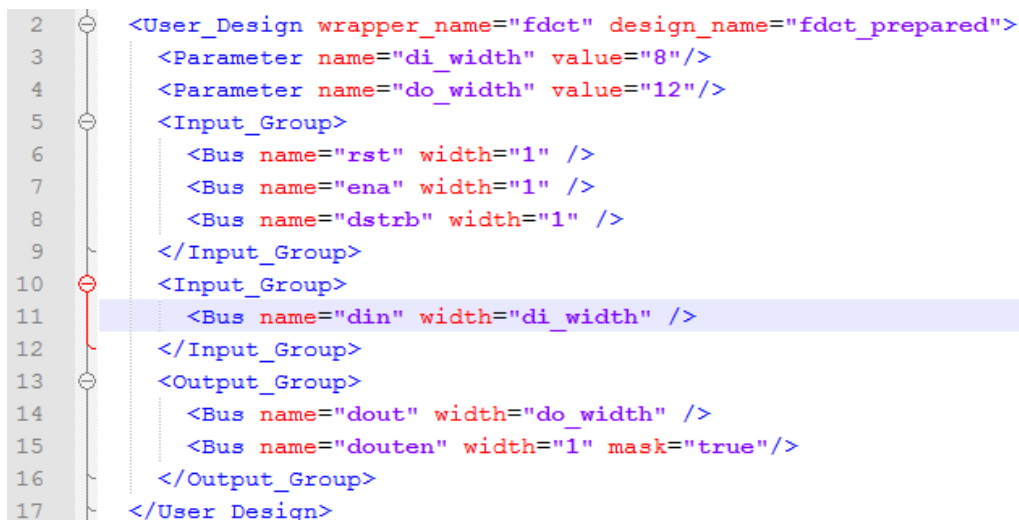
Figure 6.19: The OpenVera code is translated to microinstructions then the Serialized is generated.

The Vera language is chosen because it is a standard verification language. In future work, we are going to define a special simple language to write serializer inspired by Vera. Currently, we assume that the hardware core designer writes a test bench for his core. In the test bench, the `read_data()` function pulls input from an input data stream and the `write_result()` function pushes output on an output data stream.

6.5.3 An example for generating a serializer from a Vera description

In the following, we explain an illustrative example for generating the serializer of the DCT hardware core [69]. The DCT XML description is depicted in Figure 6.20, the Vera test bench and its translation into micro-instructions are shown in Table 6.2 and the resulted serializer is shown in Figure 6.21.

Discrete Cosine Transform (DCT) and its inverse (IDCT) are used in compressing multimedia streams in video and audio applications. DCT decomposes the signal into weighted sums of cosine harmonics. The DCT core inputs are: 8-bit data, data in strobe, reset and enable. The core outputs are 12-bit data and data out strobe. The core designer provided a Verilog testbench to explain how it works. To start computing the user reset the core through the rst input. Then, the enable input is set high. The dstrb input is set high then low, then the data is sent one byte with each clock cycle. The core receives 64 data bytes. When douten output becomes high, 64 words are captured from the dout output.



```
2 <User_Design wrapper_name="fdct" design_name="fdct_prepared">
3   <Parameter name="di_width" value="8" />
4   <Parameter name="do_width" value="12" />
5   <Input_Group>
6     <Bus name="rst" width="1" />
7     <Bus name="ena" width="1" />
8     <Bus name="dstrb" width="1" />
9   </Input_Group>
10  <Input_Group>
11    <Bus name="din" width="di_width" />
12  </Input_Group>
13  <Output_Group>
14    <Bus name="dout" width="do_width" />
15    <Bus name="douten" width="1" mask="true" />
16  </Output_Group>
17 </User_Design>
```

Figure 6.20: The FDCT I/O specification in the XML file.

Usually, the test bench should start by specifying mask and clock counter values. In line 1, the mask variable is set to 0xFF which is translated to *Set mask FF* micro-instruction. In

line 2, the clock counter variable is set to 0x00 which is translated to *Set clock counter 00* micro-instruction. Lines 4-7 assigns different values for several inputs at the same clock cycle. The tool generates one microinstruction in line 5 (i.e. *Set data* with three input values) since all of them belongs to the same input group as described in the XML specification, Figure 6.20. The micro-instruction outputs the three inputs in the internal wrapper format (i.e. CLK, SEL, DATA). Lines 8-10 also generate one similar micro-instruction. The *for loop* come next in lines 11, 12 and 15. They are translated to *LoadCounter1 31* and *LoopNZ1* micro-instructions. Only counting loops with deterministic counters are allowed. Line 13 reads an element from the input data stream which is translated to the input group index and data in the wrapper internal format (i.e. CLK, SEL, data). Lines 3 and 16 shows that the whole could is repeated for every 32 bytes. Lines 7, 10 and 14 indicates that the CLK value should be 1 for the corresponding micro-instructions.

As shown in Figure 6.21, all outputs have the wrapper internal format (CLK, SEL, DATA). Some input groups are internal (i.e. dstrb, reset and enable). Their values are determined in the test bench, not by the input bitstream. Those inputs should be placed directly at the output multiplexer since they have several constant values. The input stream (i.e. data) is placed as a separate input to the output multiplexer because of line 13 which assign it to `data_in` which represents the hardware core input.

Table 6.2: FDCT benchmark verification code written on OpenVera and its translation to microcode microinstructions.

	OpenVera Code	Microcode μ Instructions
1	Mask = 8'hFF;	00 Set mask FF
2	CLK_Counter = 8'h00;	01 Set clock counter 00
3	forever begin	
4	dstrb = 1'b1;	
5	reset = 1'b0;	02 Set data (sel=00 {dstrb,reset,enable}) = {0,1,1}
6	enable = 1'b1;	
7	#10;	
8	dstrb = 1'b0;	
9	reset = 1'b1;	03 Set data (sel=00 {dstrb,reset,enable}) = {1,0,1}
10	#10;	
11	for (i = 0; i<31; i++)	04 Load counter1 31
12	begin	
13	data_in = read_data();	05 Set data (sel=01 {data}) = data_in
14	#10;	
15	end	05 LoopNZ1 05
16	end	06 Jump 02

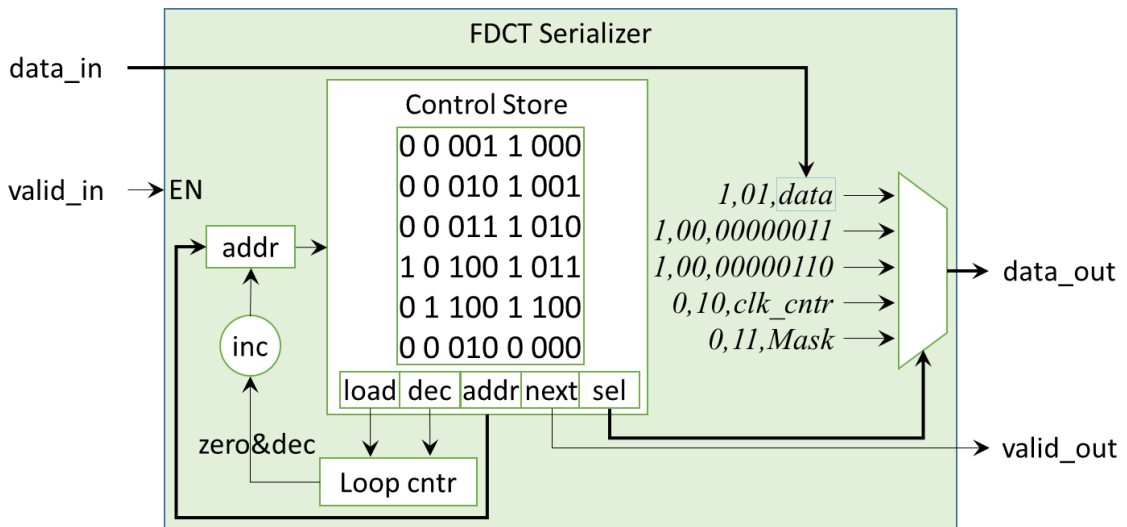


Figure 6.21: The serializer for the FDCT can be generated automatically from the Vera specification code Using a microcode-template. Each cycle of the microcode generates data for one input group and generates the group index and one bit represents whether to apply a clock or not for this data.

6.5.4 Wrapper generation software

We used MS Visual basic 6 to build the wrapper generation tool (a snapshot is shown in Figure 6.22). The tool already has a wrapper template in Verilog. It reads the hardware core specifications from the XML file (reading JSON files might be added in the next version) and generates the Verilog file of the wrapper. The generated code includes the wrapper code, an instantiation of the hardware core and instantiations of other wrapper components. The current version of the wrapper does not generate the serializer and the deserializer. The tool can modify the input/output groups and bus widths and updates the XML file.

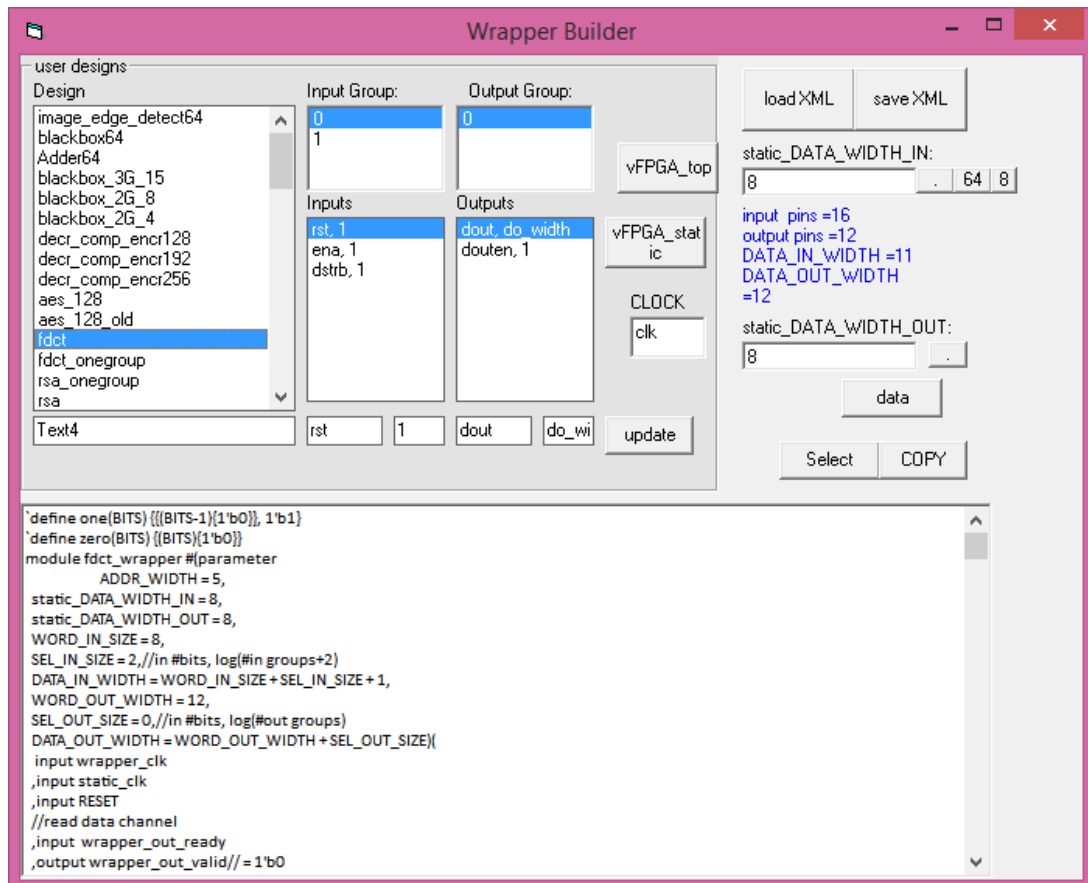


Figure 6.22: A snapshot of the wrapper builder software. The list on the left contains several hardware cores. The wrapper is generated instantly for the selected hardware core. The bottom large textbox contains the generated wrapper Verilog code.

CHAPTER 7

Results and Comparison

In this section, we first introduce a hardware core example and explain how to generate a wrapper for it and then implement it on virtual FPGA. Second, we implement the virtualization platform with four hardware core examples and evaluates the virtualization overhead in terms of area, performance, throughput, and power. A comparison between our virtualization platform and other platforms in the literature is also provided. Finally, we introduce a CCM example for the edge-detection application and show how CCM can be accessed using the same software library used to access similar software function.

7.1 Generating a wrapper for the JPEG Encode core

In the following, we show a complete CCM generation scenario for a given hardware core, the JPEG Encoder [70]. For this experiment, we use 1GE Ethernet communication with Xilinx Virtex 6 XC6vlx550t FPGA. We used the Xilinx Chipscope tool to take a running snapshot that shows how the wrapper components work and how the hardware core clock is controlled.

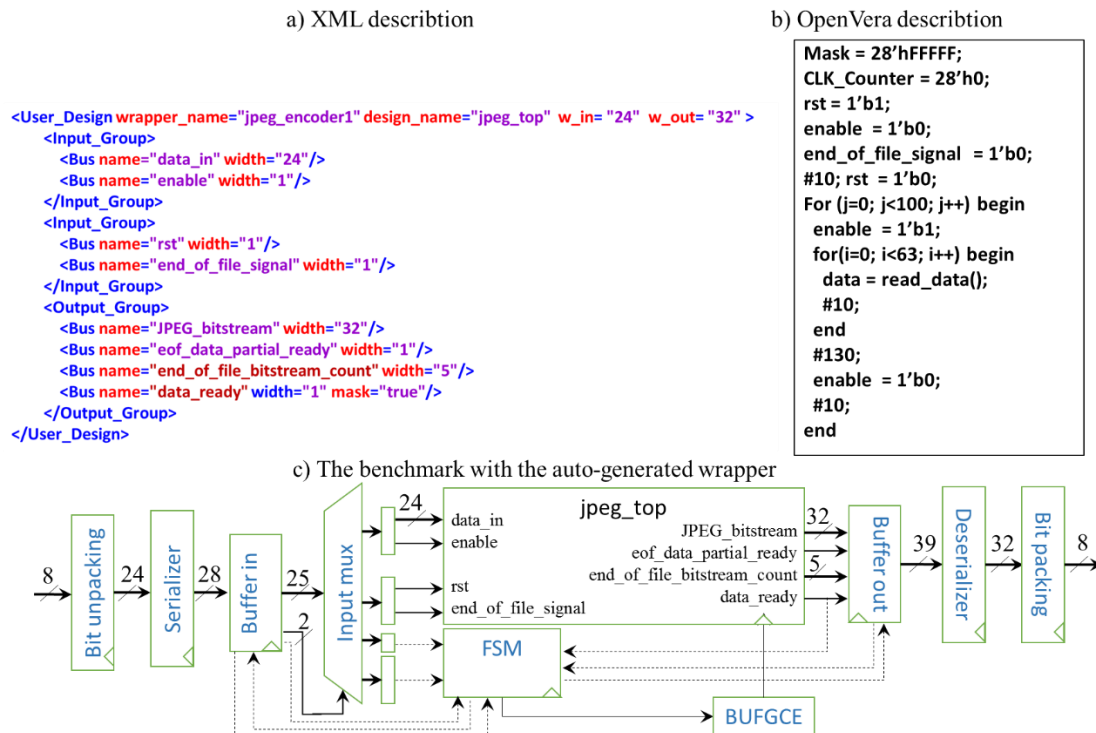


Figure 7.1: Generating the wrapper for the JPEG Encoder from the XML and Vera specifications.

7.1.1 Preparing the XML Description File

Figure 7.1 (a) shows the XML description of the JPEG Encoder's interface used to generate the wrapper. The specially developed tool assumes that the wrapper data bus width (same as the Ethernet controller's used in this experiment) is 8 bits. The XML description defines two input groups with a maximum input width is 25 bits. Adding two selection bits and one bit for clock control results in 28 bits, the input FIFO's width. For outputs, there is only one group in the XML description. Therefore, the mask register is only one bit wide,

and the tool removes the output arbiter and multiplexer since they are not needed. Setting the property mask = "true" means that the output will be captured only when "data_ready" is high. Since the total input width is 28 bits, a packing circuit is instantiated at the wrapper's input because $8 < 28$ while an unpacking circuit is instantiated at the wrapper's output because $39 > 8$.

7.1.2 User's Vera Data Specifications

Figure 7.1 (b) shows the Vera description of the JPEG Encoder's data. It describes how the data should be applied to the circuit. In this case, it specifies that the circuit receives one block of the image at a time, does computation, produces compressed data for the block, and repeat this process with other blocks until it finishes the image. Each block of data of the image is applied to the inputs using 64 consecutive clock cycles (In the core documentation 64 cycles are reported but in the simulation test bench provided by the core designer only 13 cycles are used). The "enable" signal should stay high during the input of each block and should be brought low for one cycle between every two consecutive blocks. It also specifies that there is a minimum of 33 cycles of computation between every two consecutive blocks where no new data can be applied to the inputs. The wrapper generator uses these details to generate the serializer. The generated serializer (Figure 7.1 (c)) receives 24-bit RGB color for one pixel at a time. The serializer has 28-bits output which consists of clock (1 bit), selection (2-bits), enable(1-bit), and data. The selection has two bits because the XML description defines only two input groups. The wrapper generator produces a Verilog file with the user design instantiated as a component. After generating a wrapper, a partial bitstream is generated for it and stored to be launched upon a user request.

Table 7.1 below illustrates how the generated wrapper formats and apply the received payload data to the JPEG Encoder's inputs. Each row in the table shows the 28-bit output of the serializer (which go to the input FIFO). The column with the header "CLK" shows the 1st data bit which determines whether to clock the jpeg encoder for the current data or not. The next column, "SEL", represents the next two bits that determine the index of the input register that will receive the current data. There are four options; "11" represents mask register, "10" represents the clock counter, "01" represents the second group (*reset, end_of_file_signal*), and "00" represents the first group. The next two columns are the input register's value (*EN* and *data*) if SEL="00". The rightmost column in the table is the concatenation of all these bits which represents the serializer output. The operation would start by setting the mask register (specified with SEL=11) to all 1s to read all output groups (1st row in Table 7.1). Then the clock cycle counter (specified with SEL=10) value is set to zero (2nd row of Table 7.1). After that, the reset is set to one, the design is clocked once (setting CLK=1), and then the user design is reset back (4th row of Table 7.1). Starting from the fifth data word, image blocks of 64 pixels each are applied by the serializer to the inputs as 24-bit RGB value concatenated with the "EN=1" per input cycle. At the end of the block, the clock counter is set to 13 cycles. Then the process is repeated for the next block by setting EN=0 and clock counter to zero.

Table 7.1: Formatting and applying the JPEG Encoder’s input data by the wrapper. Four control bits are added with each input. The last column represents the complete output of the serializer which is applied to the input FIFO.

	CLK	SEL		Value	Description	Serializer’s Output
	0	11		all 1’s	mask	7ffffff
	0	10		0	clock counter	4000000
	1	1		1	reset on	A000001
	0	1		0	reset off	2000000
Image block #1 (64 pixels)						
	CLK	SEL	EN	Data	Description	Data
1	1	0	1	36536E	RGB pixel	936536E
2	1	0	1	37546F	RGB pixel	937546F
3	1	0	1	47647F	RGB pixel	947647F
...						
62	1	0	1	182D48	RGB pixel	9182D48
63	1	0	1	142742	RGB pixel	9142742
	0	10	0	21	clock counter	4000021
64	1	0	1	0A1E37	RGB pixel	90A1E37
	0	10	0	0	clock counter	4000000
	1	0	0	0A1E37	EN off	80A1E37
Image block #2 (64 pixels)						
1				374862	RGB pixel	9374862
2				313F55	RGB pixel	9313F55
3				263042	RGB pixel	9263042

7.1.3 JPEG Encoder implementation on a vFPGA

The JPEG Encoder was synthesized with the generated wrapper and a partial configuration bitstream was generated targeting one of the created vFPGAs. Xilinx’s PlanAhead tool was used to make three reconfigurable regions (vFPGAs) on the FPGA beside the static logic region. The whole platform (vFPGAs, static logic, and the JPEG Encoder) was then implemented on a Xilinx Virtex 6 XC6vlx550t FPGA and the static logic’s configuration controller configured the vFPGA with the Encoder using the internal configuration port (ICAP). To verify the correctness of this implementation (i.e. platform delivers the data

correctly to the Encoder in the vFPGA), the internal signals on the FPGA were captured using Chipscope Pro Analyzer tool from Xilinx. The Chipscope clock frequency was 200MHz while the JPEG Encoder and the Ethernet network controller (with 8-bit data width) were clocked at 100MHz and 125MHz, respectively. The wrapper's input and output channels are 8-bit wide to match the Ethernet's. Hence, the packing circuit receives 8-bit data words and packs them into 28-bit data words for the serializer.

Figure 7.2 shows snapshots of Chipscope's output. Figure 7.2 shows the wrapper buses in the following order; wrapper-in, Async-FIFO-in, Async-FIFO-out, the JPEG Encoder's inputs and outputs, output FIFO-in channel, output FIFO-out channel, and wrapper-out channel. It shows how the wrapper FSM controls clocking the user design according to inputs arrival and the clocking information produced by the serializer. In addition to the packing latency, the serializer adds some delay at the beginning of each image block which requires stalling the user design according to data arrivals as shown in Figure 7.2. Therefore, the user design should receive one clock pulse each $28/8$ wrapper's cycles.

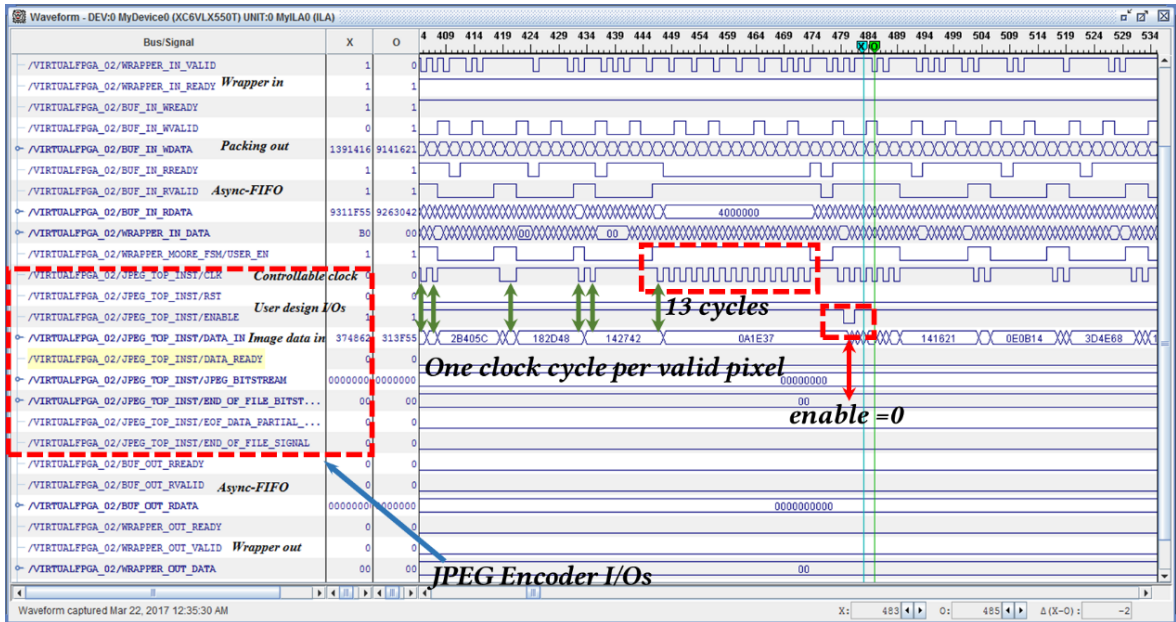


Figure 7.2: Snapshot of the complete wrapper’s and the Encoder’s input/output and control signals.

The snapshot in Figure 7.2 is taken at the end of the first 64-bit block of image data and shows the 13 clock cycles of computation followed by the enable signal (EN) going low for one cycle before the next image block is supplied to the Encoder. The snapshot also shows how data arrival is overlapped with the packing process (as indicated by the several consecutive clock cycles after the 13 computing cycles) due to the use of the FIFO at the wrapper input. The snapshots in Figure 7.3 and Figure 7.4 show the signals captured at the inputs and outputs of the vFPGA, respectively to show the data packing/unpacking process. As shown in Figure 7.3, the packing circuitry continuously packs 8-bit words into 28-bit words. Similarly, Figure 7.4 shows how the unpacking circuitry unpacks 39-bit Encoder output data into 8-bit words.

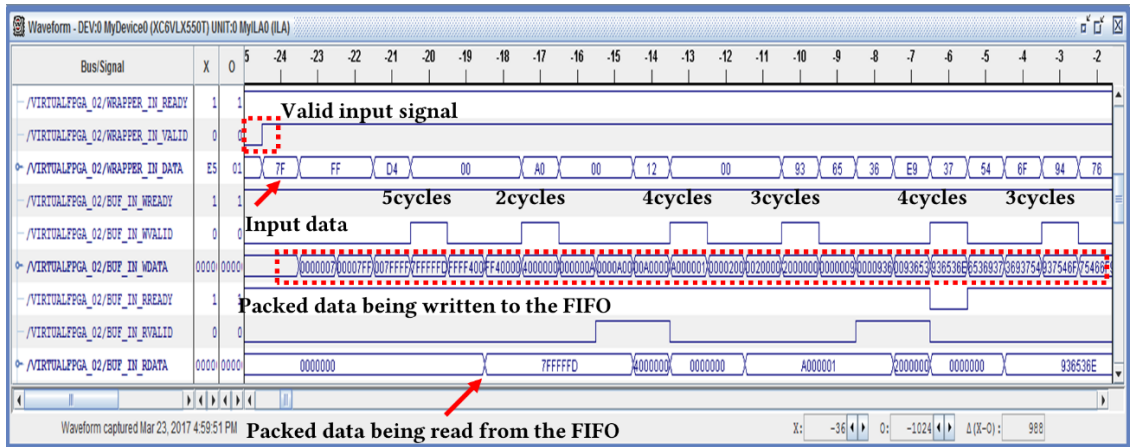


Figure 7.3: Packing the 8-bit wrapper inputs into the Encoder's 28-bit inputs in ~28/8 cycles per input. e.g. input sequence 7F FF FF D4 00 00 00 is packed into 7FFFFFFD 400000 sequence.

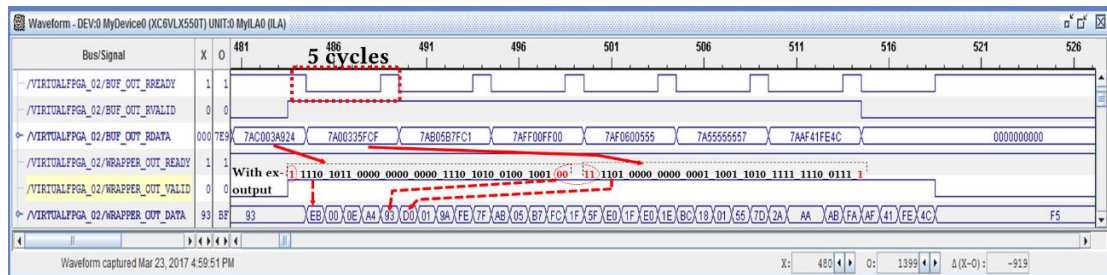


Figure 7.4: Unpacking the Encoder's 39-bit outputs to produce 8-bit wrapper's output per cycle.

7.2 Simulation methodology

Since the FPGA in our platform is an Ethernet-attached device, the whole platform simulator should receive Ethernet packet and outputs Ethernet packets. The input method of the Xilinx ISim simulator is a Verilog test bench. The process is depicted in Figure 7.5. Therefore, we need a software tool to generate Ethernet packets for the user data and a software tool to generate a Verilog test bench for these packets. In this section, we list and explain the implemented algorithms for the Ethernet packet and test bench generation. We have implemented a software tool that reads the input data (images, or encryption data), packetizes them, breaks the packets into input vectors and generates the required test bench.

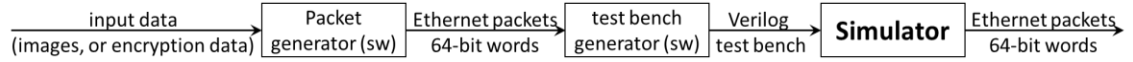


Figure 7.5: Simulation methodology to simulate the whole platform. The simulator inputs are Ethernet packets. The simulator outputs are Ethernet packets.

For each benchmark, we run two simulations. The first one is the original simulation prepared by the core builder. The second simulation is a simulation for our platform that contains the same core. We report the latency and throughput of each simulation for each benchmark.

In Figure 7.6 we show the simulation of the FDCT core as it is written by the core designer. The latency and full computation time are reported. Then, we put the core within a virtual FPGA implemented using our virtualization platform and simulate the whole platform. We use the same data used by the original simulation. We use our Ethernet packet generator to generate Ethernet packet sequences. Then, we use our test bench generation software to generate the whole platform test bench. The simulation of the FDCT within a vFPGA in our platform is depicted in Figure 7.7.

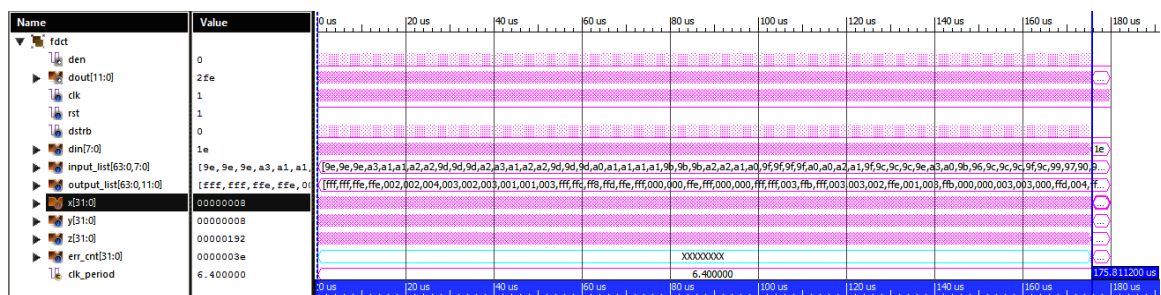


Figure 7.6: The simulation of the FDCT core as it is designed by the core designer. The total computation time is measured to be 175,811.2 nanoseconds.

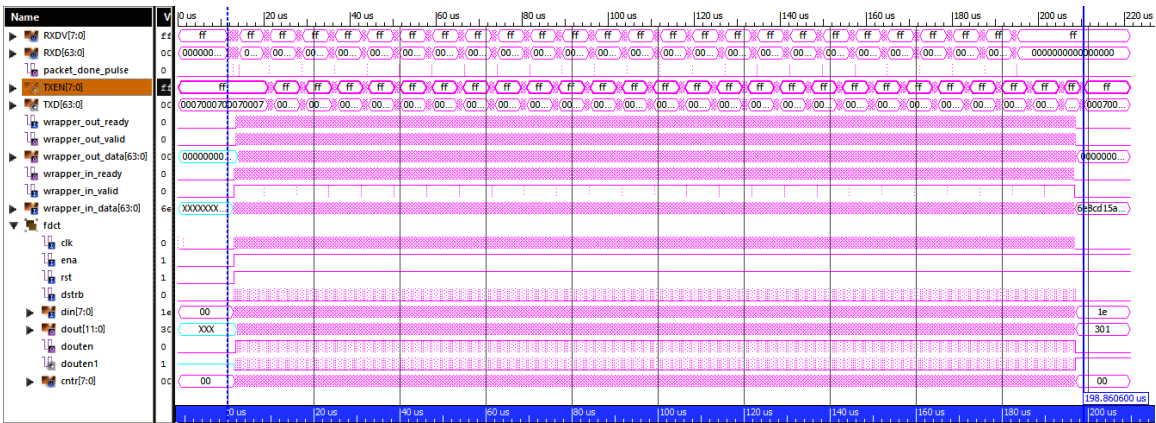


Figure 7.7: The simulation of the FDCT core placed within a vFPGA in the implemented virtualization platform (using 10GE). The time from receiving the first Ethernet packet (RXDV changes) until the last Ethernet packet is transmitted out (TXEN changes) is measured to be 198,860.6 nanoseconds.

7.3 Virtualization Overhead Evaluation

To verify the effectiveness of the proposed FPGA virtualization scheme and evaluate its area, power, and speed overhead, a complete platform was implemented and used to host four different designs placed in its vFPGAs. Four different open IP cores were used as benchmarks; an RSA512 encryption engine [71], a JPEG Encoder (JPEGEnc) [70], a fast discrete cosine transformation (FDCT) engine [69], and an AES encryption (AES128) [72] engine. A Virtex6 Xilinx FPGA with a 1/10 Gigabit Ethernet port (XC6VLX550t) was used to host the virtualization platform with four the vFPGAs. The 4 IPs were synthesized with the generated wrapper and a partial configuration bitstream was generated for each IP targeting one of the created vFPGAs. Xilinx's PlanAhead tool was used to make four reconfigurable regions (vFPGAs) on the FPGA beside the static logic and network controller regions. The 4 IP circuits were then configured on the FPGA via the static logic's configuration controller using the internal configuration access port (ICAP). Using Xilinx's ChipScope, a technology that allows real-time monitoring of internal FPGA signals, the proper operation of the wrappers was verified.

To evaluate the overhead of our virtualization scheme, it was compared to a direct implementation of the four IPs on the same FPGA (bare-metal with no virtualization) without any design modifications to the IPs. Also, to eliminate the effect of frequency on performance, all IPs for both implementations were operated at 156.25 MHz, the 10 GE Ethernet controller frequency. Though the direct implementation with inputs/outputs applied/captured directly to/from the IPs through the FPGA I/O pins may not be practical or even realizable, it constitutes the theoretical best-case in terms of area, power, and speed. That is why it was used as a baseline for evaluating the area/power/speed overhead of the proposed virtualization infrastructure.

Table 7.2 summarizes the virtualization overhead of our scheme compared to the direct implementation in terms of area, latency, power, and throughput. For these results, to obtain the overhead for each IP separately, four copies of each IP were placed on the virtualization platform since the static logic is shared between the four vFPGAs. The results in Table 7.2 are based on post place and route simulations. This is due to two reasons; (1) there is no way to inject/ readout inputs/outputs to the direct FPGA implementations, and (2) We do not have a 10 GE switch that can be used to send packets to the vFPGA platform. The total computation times are measured from sending the first Ethernet packet of the user's input data until receiving the last Ethernet packet of the results. In the case of the AES128, the computation time overhead is dominated by the communication overhead. The total computation time overhead for the other 3 IPs is acceptable because computations are more prominent than communication for these benchmarks.

Table 7.2: Virtualization overhead compared to direct implementation on an FPGA for 4 benchmarks. For the vFPGAs, the wrapper's I/O widths are 64/64 bits for all designs.

	RSA512	DCT	JpegEncoder	AES128
Inputs/Outputs Widths (bits)	64/16	14/13	28/39	128/128
Data (Bytes)				
FPGA	3,840	25,728	27,648	327,584
vFPGA	4,505	36,487	33,796	337,920
Overhead	17.32%	41.82%	22.24%	3.16%
Time (ns) using 10GE @156.25MHz				
FPGA	18,750,265	175,811	73,164	131,176
vFPGA t2	18,764,874	198,860	80,377	356,403
Overhead	0.08%	13.11%	9.86%	171.70%
Latency (ns)				
FPGA	1,249,974	439	790	131
vFPGA	1,250,151	577	941	416
Overhead	0.01%	31.44%	19.11%	217.56%
Throughput (MBytes/s) using 10GE @156.25MHz				
FPGA	0.20	139.56	360.38	2,381.60
vFPGA	0.20	123.38	328.04	876.56
Overhead	0.08%	11.59%	8.97%	63.19%
Dynamic power (mW)				
FPGA	138.91	248.64	433.25	717.71
vFPGA	362.7725	346.9925	822.1725	1239.78
Overhead	161.16%	39.56%	89.77%	72.74%
Area (Slices)				
FPGA	2,676	726	9,693	919
vFPGA	3,083	1,263	14,470	1,820
Overhead	15.21%	73.97%	49.28%	98.01%

- *Latency* was measured as the time from receiving the first input until producing the first output. For the vFPGA, the latency increase is attributed to the initialization of the mask register and the clocking counter which consumes 150~200 ns. AES128 latency increased more than others because its input size is 128 bits which is double the Ethernet data bus width of the system which in turns made the wrapper halves the IPs clock frequency to match the communication channel throughput. For such IPs

(with extra-wide input/output widths), a larger bus width would reduce the latency overhead (e.g. a 256-bit, 40 GE data-bus).

- *Average Throughput* in bits/seconds was measured as the ratio of the total data over the total time for both virtual and physical FPGAs in the table. Throughput overhead of the vFPGA platform was around 10% except for the AES128 circuit. The overhead depends on how much input/output data packing/unpacking is required and how much control bits are consumed with the data per cycle. More packing/unpacking means more time spent is preparing the data for the actual computation, increasing the overhead. Similarly, more control bits per input cycle results in less data throughput. For the RSA 512 benchmark, the IP's input width matches the wrapper's very well, hence unpacking takes very little overhead. For the DCT and JPEG Encoder, unpacking becomes more significant (the DCT is slightly better matched with the wrapper's data width). As mentioned before, due to the huge mismatch between the AES128 input width and the wrapper's, the effective frequency of this IP's clock was half that of the wrapper (and the physical FPGA version), yielding the largest throughput overhead. Again, a wider data bus would have reduced this overhead significantly.
- *Area overhead* is measured in FPGA slices and is due to the wrapper and static logic. For four vFPGA partitions, the static logic's total area is constant at 2377 slices (i.e. 9,508 LUTs, or ~ 3% of the FPGA LUTs), or ~600 slices per vFPGA. The wrapper's area dominates the area overhead and varies for each benchmark depending on its input and output size because of the packing/unpacking circuitry. The reported area of the DCT and RSA is the total area of the platform divided by four since there are

four vFPGA each of which contains one core. For the AES128 two core are implemented in two vFPGA. The reported area is the total platform area divided by two. For the JPEG Encoder, the full area of the platform is reported since only one vFPGA is implemented.

Figure 7.9 shows the XML specification of a black box (an empty design) which is used to generate different wrappers with different number of inputs and outputs and different groups. The generated wrappers are synthesized with the “keep hierarchy” option since the design is empty and the resulted area is reported. Figure 7.8 illustrates how the wrapper area (LUTS and FFs) changes as a function of the inputs/outputs data widths. For this figure, the design is treated as a black box with the equal number of inputs and outputs, and for each I/ O width, three wrappers were generated; assuming the inputs/outputs are grouped into one, two, or three groups. The wrapper's area increases exponentially with the I/O width while dividing I/Os into groups reduces the area significantly.

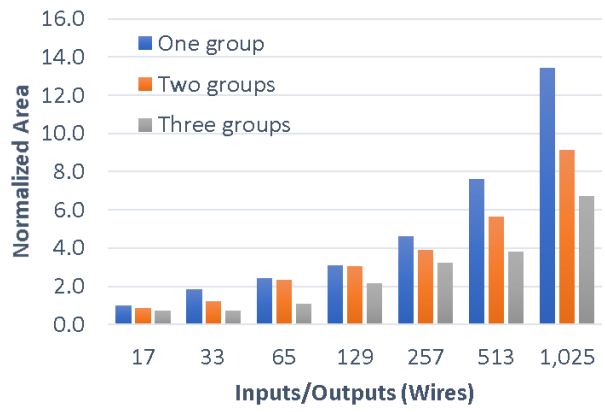


Figure 7.8: Wrapper area versus the number of the applicaion I/Os for 1, 2, 3 grouping.

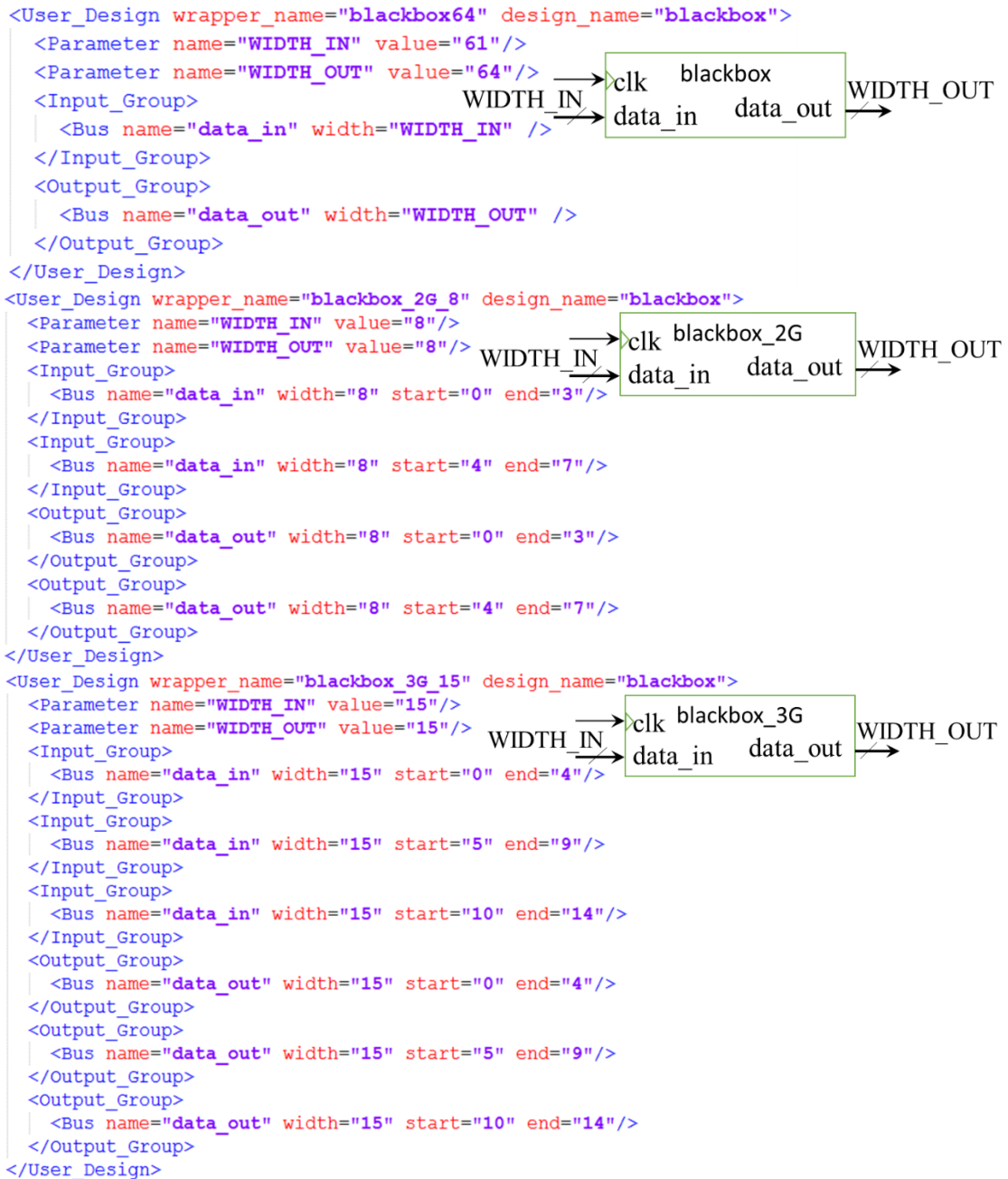


Figure 7.9: The XML specification of the input/output groups of a black box (one group, two groups, and three groups). The black box has no design inside. It is used to generate a wrapper for an assumed design with arbitrary inputs/outputs and an arbitrary number of groups. The black box is used to evaluate the wrapper area for different number of inputs/outputs and different number of groups.

- *Power overhead* is incurred due to the additional circuitry of the wrapper and static logic. The effect of the wrapper and static logic on power is more prominent for IPs that have less time overhead (e.g. the RSA512) since the total energy per computation (independent of the frequency) is spent over less time which increases the average

power. The results reported in Table 1 is based on active power (i.e. during operation of vFPGA-based designs). Also, the overhead depends on the size of vFPGA circuits relative to the wrapper and static logic. In this case, the IPs are relatively small, increasing the relative overhead. For power measurements, Xilinx's XPower Analyzer was used. It reads the placed and routed design, the physical constraints file, and net and I/O activities (from post place and route timing simulation results), and accurately estimate the power. Using post place and route timing simulations for calculating net activities includes glitches and hence results in highest accuracy.

7.4 Comparisons with other platforms

Table 7.3 shows a comparison of our FPGA virtualization platform with other notable platforms for attaching FPGAs to DCs. Many of the approaches reviewed in CHAPTER 4 could not be included in the table because they did not report area overhead or used HW macros (i.e. non-reconfigurable resources). The table summarizes the type of the platform and its interface (to the user's design), area overhead in terms of FPGA resources (for the overlay, it is reported as a ratio to the bare-metal design), the platform components (i.e. static logic), and whether partial reconfiguration is supported or not. A platform with a fixed interface means that designers must adapt their design to this fixed interface. Platforms that do not support partial re-configuration means that the whole FPGA circuitry (interface + communication + design) must be re-synthesized every time a new design is to be deployed. The VirtualRC [73] was included because it provides an abstracted application-specific interface that can be used to attach an FPGA-design to a DC. Most references do not provide performance overhead over direct FPGA implementation, so it was not included in the comparison. Platforms that provide local DDR memory access (

[6, 41, 9, 12, 38, 36]) suffer significantly higher overhead. Though this allows more design choices and applications, it adds huge overhead and thus is best implemented as hard macros. As this table shows, our proposed platform provides a complete interface abstraction and partial reconfiguration support at a comparable or less area overhead than other techniques.

Table 7.3: Comparison with notable platforms for attaching FPGAs to DCs.

Platform	Type	Area overhead	Static logic Components	PR ₊
MS Catapult [6, 41]	PCI attached, Torous network among FPGAs, PCIe DMA Specific Interface	$\approx 39,560$ ALMs _‡	two DRAM controllers, four Slite II (to connect over Ethernet), router, PCIe core, reconfiguration management	×
Disaggregated FPGAs [12, 38]	Network attached, Specific Interface (similar to OS sockets)	$\approx 58,128$ LUTs + 116,256 FFs	DRAM controller, memory virtualization module for each vFPGA network controller, management	✓
RIFFA2.1 [43]	PCIe DMA Interface	15,862 LUTs + 14,875 FFs (Xilinx) 15,182 ALUTs + 13,418 FFs (Altera) (without PCI logic)	PCIe core, tx-rx engines for 4 vFPGAs	×
DyRACT [44]	PCIe DMA Interface	16,157 LUTs + 19,453 FFs	PCIe core, tx-rx engines, reconfiguration man., clock man., DMAs	✓
Extended [36]	PCIe DMA Interface	30324 LUTs + 60648 Regs	added DRAM communication & interface	✓

RACOS [45]	PCIe DMA Interface	7474 LUTs + 7466 Regs	PCIe core + DMA bus mastering component + Register file + Reconfiguration management + Event dispatcher + two 64-bit wide FIFOs per vFPGA	✓
Byma [9]	Network Attached, DPR*, Specific Interface for Packet Processing Applications	28,711 LUTs + 29,327 FFs	Soft processor (Reconfiguration management), DRAM controller, MAC Regs., Mem mapping Regs.	✓
VirtualRC [73]	Domain Specific with Specific Interface	2,300 LUTs + 4,550 FFs	N/A	✗
This Work	DPR, General	9508 LUTs+4344 FFs (4 vFPGAs)	Network controller (complete TCP stack), clock management, reconfiguration management	✓

+ PR=Partial Reconfiguration Support

‡ ALM= Adaptive Logic Module (Altera), equivalent to Xilinx's Slice (6-input LUT + 4FFs).

* DPR= Dynamic Partial RE-Configuration (for vFPGAs).

7.5 vFPGAs versus SW-based virtual machines

Cloud-based applications usually run on virtual machines or within containers which introduce remarkable overhead compared to running the same application on the physical machine. To show the viability of FPGA-based computing in clouds with our proposed vFPGA platform, the performance of an actual streamed application (not simulated) is evaluated when it is run on a virtual machine, a physical machine, and on a vFPGA, all in an environment similar to a cloud's. The purpose of this experiment is to show that FPGA-based streamed applications do not lose their speed advantage over SW implementation even when the FPGA is virtualized using our proposed methodology. For this experiment, we designed a custom streamed application that we believe is a good representation of applications that are suited for both, cloud environment and FPGA implementation. The

application involves three main sequential tasks performed on streamed blocks of data; decrypt-compute-encrypt, i.e. it receives encrypted data, decrypts it, performs some relatively simple computation on the plain text, then encrypts the results and send them back to the user. Symmetric key encryption (AES) was used for the encryption and decryption tasks. For the three application platforms, a client application (running on a typical workstation) streams the data over a 1 GE LAN to the three different platforms and receives the streamed results back as illustrated in Figure 7.10.

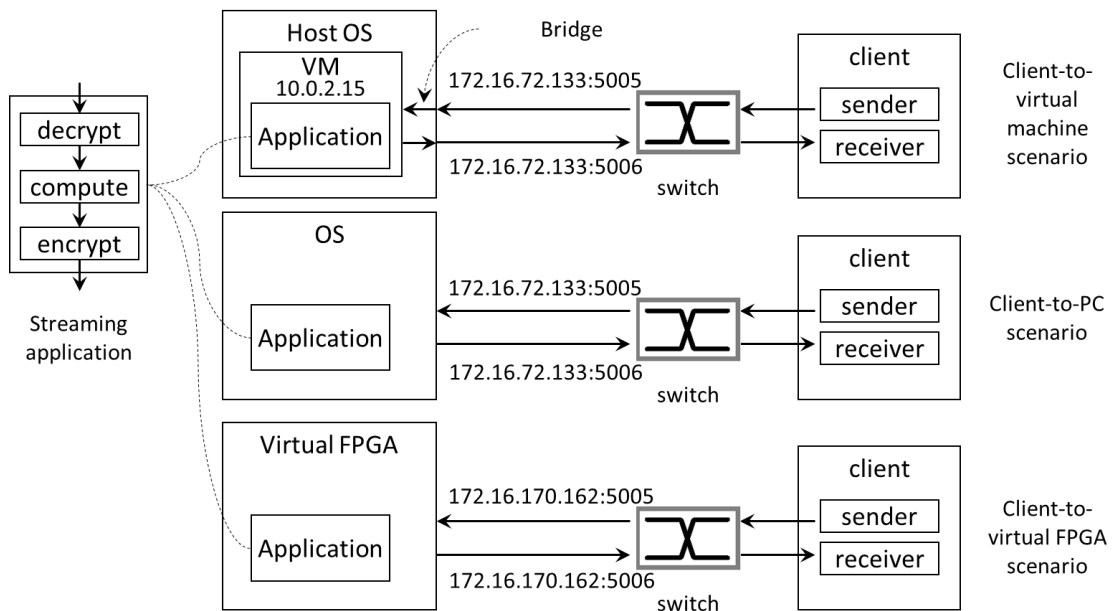


Figure 7.10: The three platforms used to evaluate the performance of a streaming application; (a) Running in a virtual machine, (b) directly on the physical server, and (c) on a vFPGA. A client SW sends encrypted data and receives encrypted computation results.

In the client-to-physical server scenario, the application was run (as a server) on a Xeon machine with 8 cores running at 3.00 GHz, 16 GB of RAM, and 64bit-Linux Ubuntu 16.04LTS. In the client-to-virtual machine scenario, VirtualBox was used to build a virtual machine with 4 GB RAM and 64bit-Linux Ubuntu 16.04LTS on another Xeon machine with the same specifications as the first one. The application was written with Python using the Python stream socket programming [74] and the Python Cryptography Toolkit

(PyCrypto) [75]. The measured stream socket throughput between two machines using our code was 115 Mb/s which represents 90% of the 1GE link theoretical bandwidth.

The hardware version of the application was built using Hsing's AES core [72]. Since Hsing's core only provides AES-ECB mode encryption, it was modified to implement AES-CTR (for encryption and decryption) which provides stronger security. Two separate instances of the AES-CTR core are used to decrypt and encrypt the streamed data. All the three platforms utilized TCP streams to/from the client over the 1 GE LAN switch with a measured sustainable throughput of ~115 Mb/s.

The application's performance was evaluated using the measured throughput as a function of the streamed block size for the three implementations as shown in Figure 7.11. The total data size was 32 Mbytes and the block size was varied from 16-bytes to 1 KB. Figure 7.11 shows that the throughput of all platforms is affected by the data block size but starts to saturate beyond a block size of 128 Bytes. In the client-to-physical server scenario, the maximum attained throughput was 29.5 MB/s while the virtual machine's version maxed out at 7.4 Mb/s. However, the vFPGA version reached 105 Mb/s, approaching the communication link's measured maximum throughput (shown on the graph). In fact, the vFPGA version throughput was limited by the communication link's throughput not the computation speed as the maximum frequency of the circuit (post place and route) was ~ 378MHz. Had a 10 GE was used, the AES128 throughput would have been 876.6 Mb/s as was shown in Table 7.2.

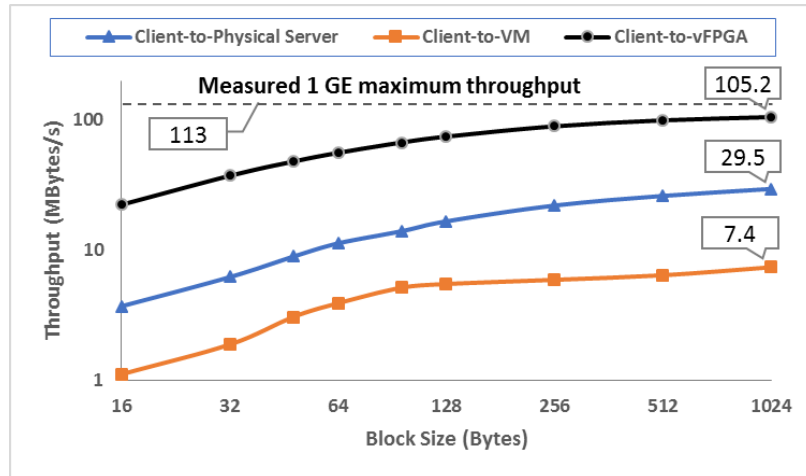


Figure 7.11: Streaming application throughput versus block size comparisons the proposed vFPGA platform and physical servers and virtual machines.

7.6 CCM platform Evaluation

In this section, we use the “image edge detection” application as a test case and show how the CCM of this application can be accessed as a service using the same functions used to access the software version. The test case is an “image edge detector” that receives a JPEG image and produces another JPEG image of the detected edges. To this end, we built a CCM for the application as well as the FPGA virtualization platform. Then, we wrote the application in software using standard python libraries and used the TCP stream socket to build the software application interface. The software version of the application running on a server and another copy is running on a virtual machine. We also designed another software to act as a user that requests the application service. The user uses TCP stream socket interface to request the service from CCM or the application. We also analyze the virtual FPGA booting time.

7.6.1 Experiment setup

The experiment setup, as shown in Figure 7.12, consists of an FPGA, a workstation represents a server, another workstation holds a virtual machine, one all-in-one machine represents the user and Ethernet switch that links those devices. The FPGA is a Xilinx

Virtex 6 XC6vlx550t FPGA. The server machine is a Dell WorkStation with Intel 8-core Xeon processor running at 3.00GHz, 16GB of RAM, and 64bit-Linux Ubuntu 16.04LTS. The VM machine is a VirtualBox virtual machine with 4 GB RAM, bridged Ethernet and 64bit-Linux Ubuntu 16.04LTS on Dell WorkStation with same specifications. The user machine is a Lenovo all-in-one machine with Intel Core-i7 processor running at 3.00GHz, 16GB of RAM, and 64bit-windows 8.

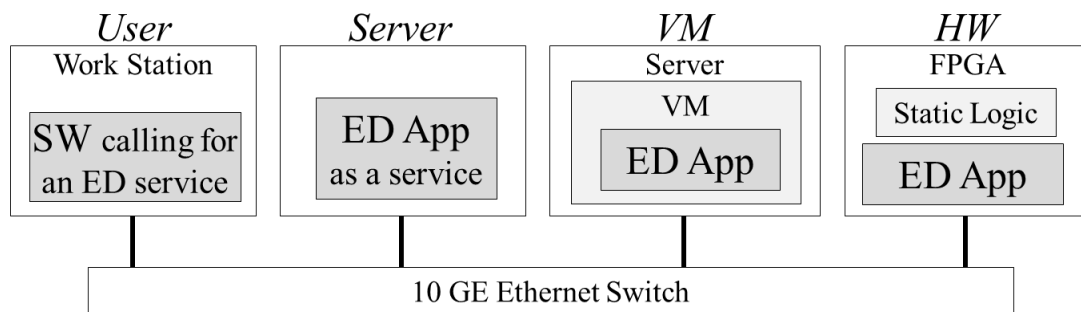


Figure 7.12: The experimental setup with several versions of the secure edge detection (ED) application.

Sender and receiver are two separate threads running in parallel on the user’s workstations. It is not possible to use one thread that sends data while it is listening to a TCP port for the received results. This better to be done using two separate threads. It is difficult to have a timer that starts by the sender and stops by the receiver. Instead, we do a synchronization process (illustrated in Figure 7.13) before starting and let server, sender, and receiver start at the same time. We put the timer in the server since it is in the in the middle. The server sends a small message (acknowledge) to sender and receiver to indicates starting the process and the timer.

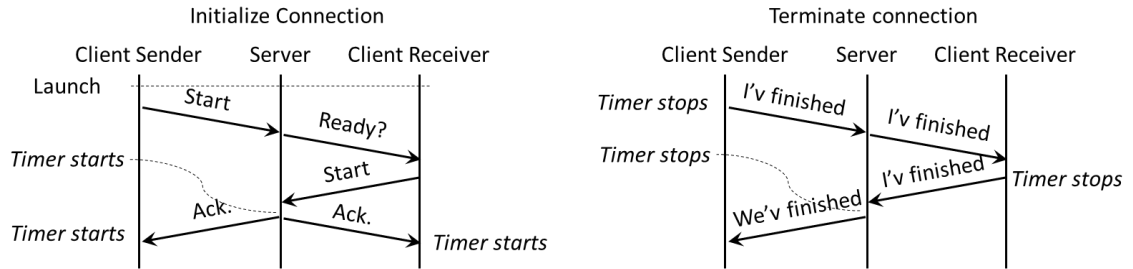


Figure 7.13: Synchronization process among the sender, receiver and the server. The server manages to start and to end the work in the three steps at the same time.

7.6.1.1 The CCM service

The virtualization platform shown in Fig.8 is implemented on the Virtex 6 FPGA. The Static logic components are all written using Verilog except the encryptor/decryptor. The tinyAES [72] is used to build counter mode AES encryption/decryption blocks (AES-CTR) explained later. The test case hardware is designed by collecting several cores; Jpeg decompressor [76, 77], Canny edge detector [78] and jpeg encoder [70] to form the edge detector hardware as shown in Figure 7.14. Then, the wrapper is formed to form the CCM and integrated with the static logic.

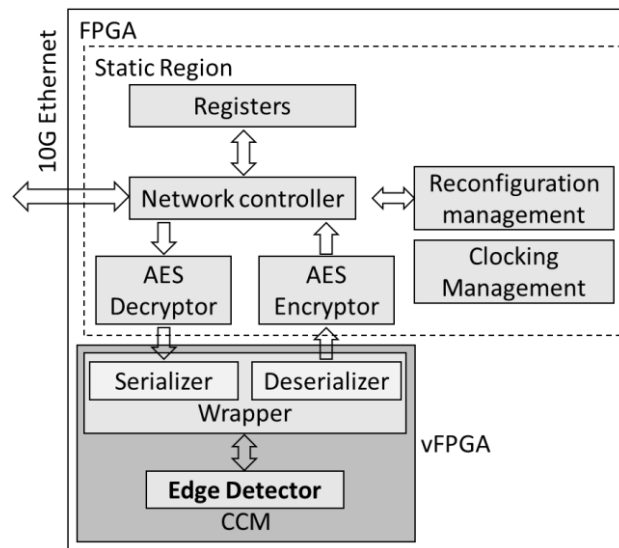


Figure 7.14: The FPGA virtualization platform with the Edge detector application implemented as a CCM.

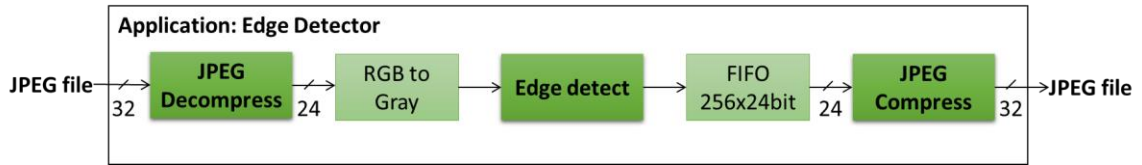


Figure 7.15: Image edge detection hardware uses four already-made cores; AES 128 [72], Image Compress [69], Canny Edge Detection [78], and JPEG encoder [70].

The hardware version of the image edge detection is shown in Figure 7.15. The hardware phases are overlapped to get high performance. The total execution time is dominated by the jpeg decompression time. Jpeg encoder has a fixed time per pixel. The decompressor time varies according to the input image.

The resource utilization report is shown in Table 7.4. The static logic uses contains the 10GE network controller and the clock management. The user hardware contains the secured image edge detector with its wrapper.

Table 7.4: Resource utilization of the virtualization platform on FPGA

	LUTs	FFs	RAMs	DSPs
Total	58123	52649	422	560
Static logic	12462	10990	161	0
User Hardware	45661	41659	261	560
- decode	11457	8428	9	21
- detect	3262	3833	18	0
- encode	26761	29966	30	560

7.6.1.2 The software service

The software of the secured image edge detection is written on Python and launched on the server and on the virtual machine. To ensure the best software throughput, we use standard libraries to build the application which proves high throughput; the standard Python Cryptography Toolkit (PyCrypto) [75] that provides encryption and decryption services and the open source computer vision (OpenCV) for python [79] that provides cany edge detection. A snapshot of the application service python code is shown in Figure 7.16. The service first decrypts the received Jpeg image, stores it in an array to pass it to the image

decoder. Then, Canny edge detect function from OpenCV library detects the edges and the resulted image is encoded again to produce a Jpeg image. Finally, the resulted Jpeg image is encrypted to be sent to the user.

```
#Server side (software compute node)
from Crypto.Cipher import AES
import numpy
import cv2
mode = Crypto.Cipher.AES.MODE_CTR
ctr_encr=Crypto.Util.Counter.new(128,initial_value=long(variables.IV.encode("hex")),16)
ctr_decr=Crypto.Util.Counter.new(128,initial_value=long(variables.IV.encode("hex")),16)
AES_encr=Crypto.Cipher.AES.new(variables.key, mode, counter=ctr_encr)
AES_decr=Crypto.Cipher.AES.new(variables.key, mode, counter=ctr_decr)
...
def compute(data_in):
    img      = AES_decr.decrypt(data_in)
    nparr    = numpy.fromstring(img, numpy.uint8)
    img_np   = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
    edges    = cv2.Canny(img_np,100,200)
    img_np2  = cv2.imencode(".jpg", edges)
    return AES_decr.encrypt(img_np2[1].tostring())
```

Figure 7.16: The software version of the application “Secure image edge detection” written in Python using standard SW libraries

7.6.2 Performance Evaluation

Virtual machines suffer from large overhead which makes them unsuitable for remote or on-cloud computations. A comparison is done between the virtual machine and CCM to reflect the strength of the CCM cloud for streamed-data applications. Figure 7.17 shows three scenarios of computation; a) uses a virtual machine, b) uses a server, c) uses CCM. The network on all cases is 10GE LAN network. The client sends encrypted images and receives encrypted edge-detected images over TCP stream sessions. The TCP stream socket on python proves high throughput approaches the theoretical line throughput.

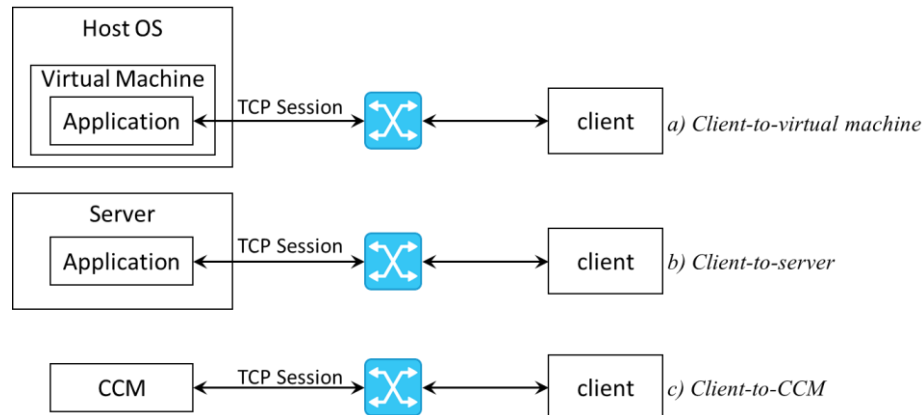


Figure 7.17: The user uses the same socket interface to request the same service hosted in three different machines; a) the service is hosted in a VM, b) the service is hosted in a server, c) the service is a CCM on virtual FPGA.

We have prepared 10 jpeg images with the same dimensions but have different sizes. The variation in size reflects the compression ratio which varies according to the image contents. Each image is encrypted using AES128-CTR and sent to the edge-detection service over the socket interface. The edge-detection service decrypts the image, decodes it, does edge detection, encodes the detected-edge image, encrypts the resulted image and returns it to the sender. Each image is sent 100 times and the average time is reported in milliseconds. The three scenarios shown in Figure 7.17 are implemented and a comparison among them is shown in Table 7.5. The performance of the service on a VM is severely affected by the VM virtualization overhead. By comparing the bare-metal server with the virtual machine, we can see that the virtualization overhead decreases the performance by 50%. The table also shows that CCM can achieve 3~4x better performance than virtual machines for this application.

Table 7.5: Computation time comparison for three implementations of the secure image edge detection application. a) The application on a virtual machine, b) The application on a server, c) The application is a CCM.

Filename	Size (Bytes)	Frames per seconds		
		HW	VM	Server
0009_640x480.jpg	58,962	232.41	57.54	103.37
0002_640x480.jpg	72,618	209.03	48.58	94.02
0004_640x480.jpg	84,644	197.17	45.13	83.33
0005_640x480.jpg	116,391	140.45	38.88	67.88
0000_640x480.jpg	128,573	117.65	28.50	65.54
0008_640x480.jpg	163,301	99.49	29.89	53.44
0006_640x480.jpg	195,211	86.32	28.35	50.08
0007_640x480.jpg	201,071	81.45	27.59	50.17
0001_640x480.jpg	266,529	71.13	22.11	42.52
0012_1920x1080.jpg	864,475	20.00	4.84	10.09

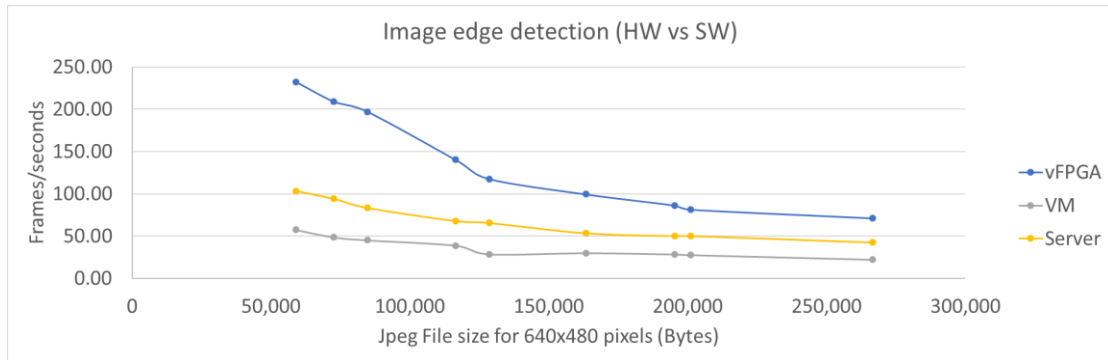


Figure 7.18: Compute nodes performance comparison for a specific application. vFPGA outperforms a virtual machine and a bare-metal server.

7.6.3 The impact of adding the AES encryption/decryption

Using the AES-ECB encryption core [72], we have built the AES-CTR which can be used as an encryptor and decryptor at the same time. In the hardware implementation, we have to have two separate instances of the AES-CTR for encryption and decryption. Figure 7.19 shows how we have built the AES-CTR from AES-ECB core. The circuit is simple and provides better throughput and stronger encryption [80].

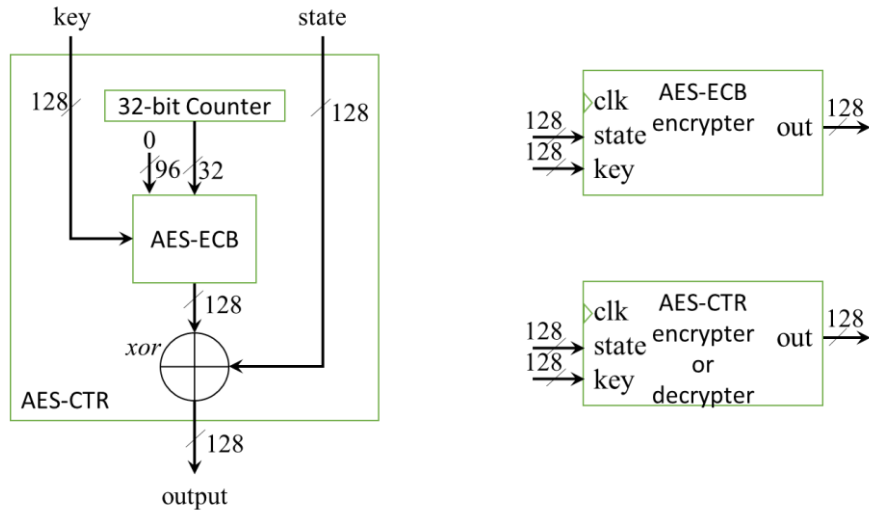


Figure 7.19: Using the AES-ECB core [72] to build AES-CTR that can be used as a decrypter and encrypter. By XORing the input text with the encrypted counter output we achieve a throughput of one data block per cycle. AES-CTR throughput is one block per cycle because the XORing takes one cycle only.

- *Latency:* To start the AES-CTR the counter should be initialized and enabled. Since the original AES-ECB latency is 20 cycles, the AES-ECB starts encrypting for 20 cycles. When the initialization is done, the core is enabled when there is incoming data
- *Throughput:* The core produces one output per cycle which means, the throughput is not affected by the inclusion of AES. This is because the critical path of the AES-CTR consists only of the XORing logic.
- *Area:* AES-CTR requires BRAMs, LUTs, and FFs which increases the static logic area. Table 7.6 illustrates the required static logic area for the AES encrypter/decrypter. It also includes an estimation (from a real commercial implementation [65]) for the full network controller stack assuming four TCP engines are implemented within the FPGA. The total area is ~60k LUTs.

Table 7.6: Estimation of the CCM area overhead in our platform.

	LUTs	FFs	RAMs
Full stack (UDP and 2 TCP engines) [65]	20,000	20,000	
Additional 2 TCP engines [65]	30,000	30,000	
AES encrypter	3,536	3,968	86
AES decrypter	3,536	3,968	86
Other static logic components	5,390	3,054	3
Total	62,462	60,990	175

7.6.4 The impact of having multiple vFPGAs within the same FPGA

The static logic routes the received data to the corresponding vFPGA. It transmits all the results of all vFPGA. In the first case, when receiving a packet, the packet is forwarded to the corresponding vFPGA directly. This process is not affected and does not affect other vFPGAs. There is a double buffer in the network controller that allows receiving a packet while forwarding a new one. If the vFPGA is not ready to receive new data, the received packet will be thrown and a retransmit is required. This way we keep the routing overhead of the received packet at a minimum level.

For the transmission part, the router receives results from all vFPGA in a round-robin manner. A transmission happens when the router receives a packet payload (1400 bytes) or when the timer time-out, then it moves to the next vFPGA. The timers job is to prevent any deadlock caused by waiting for results from a specific vFPGA. If we have n vFPGA sharing the same transmitter and producing results at the same time, their throughput will be affected by $1/n$ at most. A good solution for this issue is to have n Ethernet plugs with and n network controllers. Each controller serves one FPGA only. In this case, the routing logic in the static logic is completely removed.

7.7 Boot time analysis

The vFPGA-CCM boot time is measured from the time the user sends a “Launch a CCM” request to the time the user receives a response with the launched CCM IP Address as illustrated in Figure 7.20. The components of this delay are:

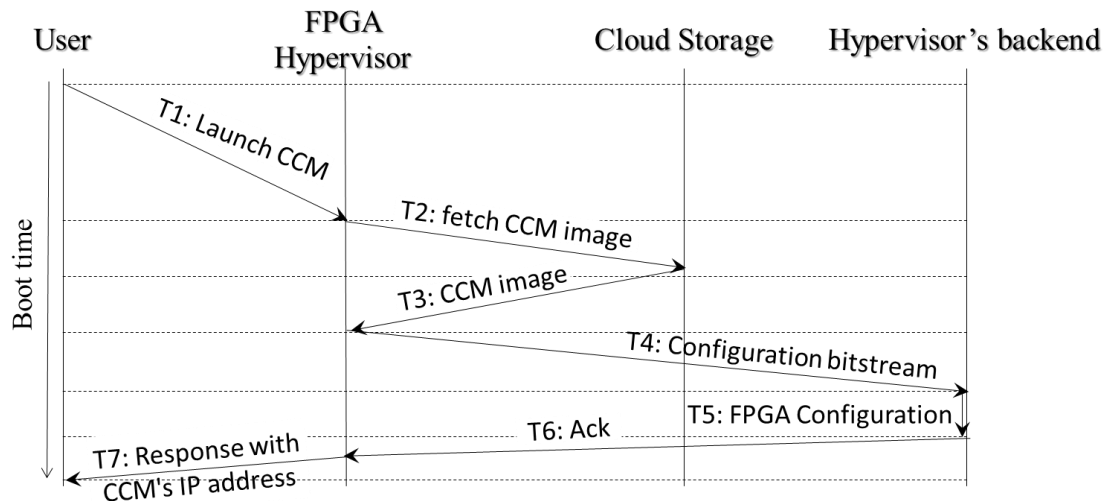


Figure 7.20: Different boot time components of a vFPGA-CCM.

- 1) Message passing delays (e.g. the launch request, the response with the IP address, etc.). This delay can be approximated to the ping time between the user and the FPGA hypervisor in the cloud. The average estimated ping time in amazon web services around the world is ~250ms as measured using the cloud info web site [81].
- 2) Fetch the CCM image from the cloud storage and sending it to the FPGA hypervisor's back-end. This delay depends on the internal network throughput within the cloud network. In our experiment setup, we have a LAN with 1G switch and the measured throughput reached 112 megabytes per second.
- 3) The FPGA configuration time by the hypervisor's back-end. Large FPGAs have an average bitstream file size of ~10 megabytes, resulting in ~25 milliseconds average

configuration time through the internal configuration access port (ICAP) which has a configuration bandwidth equals to 3.2 Gbps [82].

Table 7.7 shows the different delay components and the total vFPGA-CCM boot times for several sizes of CCM image (i.e. bitstream files) sizes. Comparing to VM booting time, the only difference is the configuration time.

Table 7.7: Boot time delay components for vFPGA-CCMs with various image (bitstream) sizes. Internal configuration access port's speed is ~400MB/s

Bitstream size (MB)	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5: Configuration time (ms)	T6 (ms)	T7 (ms)	Total Boot time (ms)
1	250	5	9	9	3	5	250	530
5			44	44	13			611
10			89	89	25			713
15			133	133	38			814
20			178	178	50			916

CHAPTER 8

Conclusion

In this dissertation, we introduced the FPGA-based custom computing machine (CCM). It is highly abstracted application hardware which can be used through software functions by users with a non-hardware background. We also introduced a cloud platform that manages FPGA resources and provides CCM-as-a-service. The introduced cloud platform can be integrated with existing data-centers and cloud platforms, provide its services, and uses their cloud resources without deep modifications on those cloud platforms. Existing works on using FPGA for doing computations focuses mainly on using PCIe-attached FPGAs as accelerators. We introduced a network-attached FPGAs and virtualizing them as network-attached standalone compute machines.

We introduced our new FPGA virtualization platform which consists of several abstraction layers that abstract the CCM to the level that it accepts data through socket communication in their original structure without control information. A physical FPGA is partitioned into static logic and partially reconfigurable regions representing vFPGAs. An abstract interface between static logic and the vFPGAs has been developed in a form of an automatically generated wrapper. This allows users to place any circuit IP in the vFPGA, send, and receive data from their IP through standard Ethernet communication. The virtualization platform is evaluated and

the overhead is reported in terms of area, performance, throughput, and power with several hardware cores.

We explained the application wrapper; a circuit that is auto-generated for any hardware core and enables it to fit in virtual FPGAs. We explained the wrapper components and the required specification files to generate it. We introduced a test case, explained the steps of generating its wrapper, implemented it on virtual FPGA and evaluated it. The wrapper represents an area and performance overhead. We provided an analysis that shows that the wrapper area is related to the input/output of the design not by the design size. This means that the wrapper area can be controlled by careful design of the serializer and define inputs and outputs of the hardware core.

Comparison with other platforms for attaching FPGAs to DCs showed that the area overhead of our proposed platform is within the same range of others but with the added advantages of having an abstract interface, support for partial reconfiguration, and not being domain specific. Comparison with SW based cloud implementations showed that our platform is a very viable computing option in the cloud.

8.1 Platform Limitations

This virtualization platform presented in this work has several limitations:

- The platform targets introducing standalone FPGA custom computing machines. It does not targets introducing FPGA as accelerators. In acceleration, there is a software/hardware partitioning process and FPGA should be attached and managed by a closed server. Using our platform in acceleration is not preferable since PCIe-attached FPGAs have faster and dedicated communications with the server. Our platform is good for streaming applications not for acceleration.

- This platform does not use off-chip DDRAMs. The memory in our platform is limited. It follows the streaming computing model where inputs are presented as a sequence of items. Unlike the acceleration model where all inputs can be stored in an off-chip memory before doing the computations.
- The platform is not suitable for packet processing systems. The CCM is not aimed to be used to accelerate software-defined networking (SDN) functionalities or for load balancing depending on Ethernet packet information. This is because the CCM receives the payload of the packets, not the packets themselves.
- The platform depends on and consumes a lot of clock resources. Each user design work on a different clock than the wrapper. The static logic has its own clock. The network controller has two clocks for receiving and transmitting. For example, a platform with four virtual FPGA requires 11 clock domains. This adds complexities to the place and routing phase and may result in un-routable designs. In modern FPGAs, there are enough resources to do this. We also suggest building new FPGA architectures that support virtualization.

8.2 Future work

The presented platform is still in its initial version. There are some advanced steps can be done to improve it:

- The platform is based on a layered approach. We have distinguished four abstraction layers and clearly identified the interfaces among these layers. The next step is to move some layers as external chip and utilize more FPGA resources for applications. For example, the network controller can be made completely off-chip. In current FPGAs, the physical layer of the network controller is presented as a small chip with

the (10-gigabit media-independent interface (XGMII). We suggest introducing the complete network controller with a full TCP stack as a standalone chip (ASIC or small FPGA).

- New FPGA architecture can be inspired. Some parts of the static logic could be made as hardware macros. The virtual FPGA is a dynamically configurable region which provides specific FPGA elements as computing resources. Other resources such as clock buffers and oscillators are not available as parts of the vFPGA regions. Several static components could be made as hardware macro in the FPGA and build new FPGA with more computational resources. Using FPGA for computation may lead to new FPGA architecture that focuses on providing extendable reconfigurable regions.
- We may build a hardware emulator using the virtualization platform. By removing the serializer from the wrapper and introducing it as a tool at the user side. The serializer software tool can read data files, translates them according to the Vera description file and produces data using the internal wrapper format. The Vera description, in this case, represents the test benches of the user hardware to be run in the remote emulator.
- The wrapper generator software tool should be developed to introduce more customized wrappers. We may need to improve the Verilog templates to provides several designing choices for the wrapper FSM, the packing/unpacking circuits and the serializer. The tool may instantly generate area and performance overhead estimations to help the designer to provide a CCM with specific specifications.
- We need to evaluate using vFPGA chain using a large application that can be divided into several phases. We can compare this with Microsoft catapult that is used to

accelerate the Bing search engine. Building a large application using several CCMs clostured as a vFPGA chain provides faster design time than combining all cores on one CCM.

Appendices

A. Description of the used Benchmarks

Several already-made open-source cores are used to test and evaluate our platform. In this section, we provide descriptions of these core with their input and outputs as written by their designers. We show the XML specification we wrote for each core to define its input and output groups.

The cores are chosen randomly. They are not meant to be powerful cores with powerful performance. We were just looking for

any open source core with clear functionality and have complete design files with a good test bench. The purpose is to generate a wrapper for different already designed cores.

We have also slightly modified the interfaces of some cores, as explained below, to make dealing with them easier. This, of course, enhanced the throughput overhead of our platform since it can minimize the switching among input or output groups at run-time.

A.1. JPEG Encoder Core

The jpeg encoder core [70] receives bitmap image, compresses it, and outputs JPEG image. The input bitmap image consists of sequences of 24-bit words which represents the RGB color of a pixel. The output of the core is sequences of 32-bit words represents the JPEG data. The other inputs and outputs of the core are control signals. The actual inputs and outputs of the core are depicted in Figure 8.1. The figure also shows an XML description for a defined input and output groups for that core. There are two input groups. The first input group is the 24-bit RGB data with the enable signal that works as a strobe signal. The second input group contains the control signals that are not frequently used. The reset signal

is used when a new image starts and the end_of_file_signal is used to indicates the last image block. There is only one output group contains the 32-bit JPEG data and the other output control signals. The data_ready works as a strobe to indicate the valid 32-word output. The eof_data_partial_ready indicates that only part of the 32-bit word should be taken. The end_of_file_bitstream_count determine how many bits should be taken from the last 32-bit word.

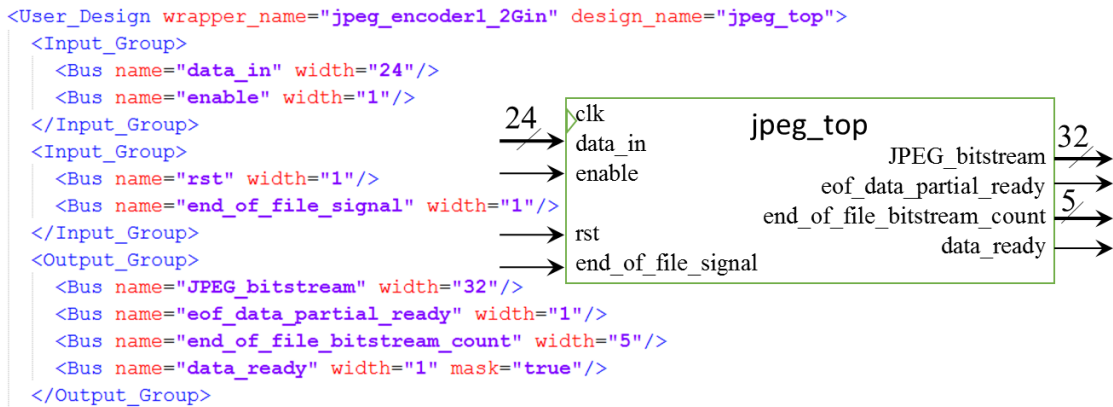


Figure 8.1: The XML specification of the input/output groups of the jpeg encoder core [70].

Table 8.1: Snapshot of the serializer’s output for the JPEG encoder. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register and the clock counter. Then it sets values to the least frequently used signals. After that, the data starts.

CLK	SEL	25-bit data	28-bit serializer’s output	Description
0	11	1 111111h	7fffffff	mask
0	10	0 000000h	4000000	clock counter
1	01	1 111111h	A000001	reset on
0	01	0 000000h	2000000	reset off
1	00	1 36536Eh	936536E	RGB pixel
1	00	1 37546Fh	937546F	RGB pixel
1	00	1 47647Fh	947647F	RGB pixel

A.2. AES Core

The Advanced Encryption Standard (AES) core [72] has two inputs; 128-bit for the encryption key and 128-bit for the data block. It has one 128-bit output. The core starts the

encryption once the key or the data is changed. The encryption process takes 20 cycles. The core is pipelined and produces continuous output while assuming changes on the input on each cycle. To control this process, we decide to add valid signals at the input and the output of the core. The new top-level module of the AES128 Verilog code is as follow:

```

module aes_128_prepared (input clk, valid_in, [127:0] state, key,
output valid_out, [127:0] out);
  aes_128 aes_128_inst (.clk(clk), .state(state), .key(key), .out(out));
  assign valid_out = cntr[0];
  reg [20:0] cntr = 21'd0;//shift_reg
  always @(posedge clk)
    cntr <= {valid_in, cntr[20:1]};
endmodule

```

The actual inputs and outputs of the core are depicted in Figure 8.2. The figure also shows an XML description for a defined input and output groups for that core. There are two input groups. One for the 128-bit data with the valid signal and the other for the 128-bit key which changes less frequently. There is one output group contains the 128-bit output with the valid signal.

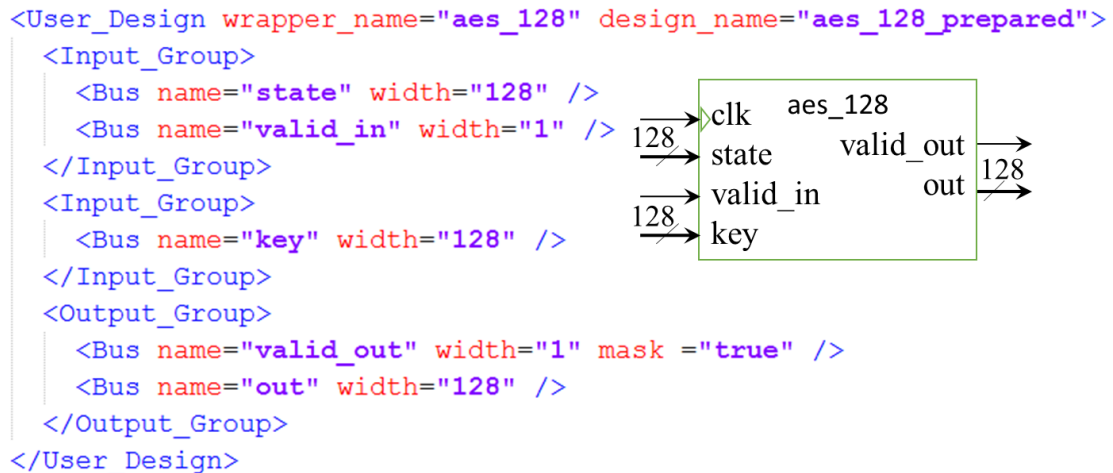


Figure 8.2: The XML specification of the input/output groups of the jpeg encoder core [72].

Table 8.2: Snapshot of the serializer’s output for the AES128. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register and the clock counter. Then it sets the key. After that, the data starts.

CLK	SEL	129-bit data		Description
0	11	1	ffffffffffffffffffffffffffffffff	mask
0	10	0	00000000000000000000000000000000	clock counter
1	01	0	76af95972db498a82052e1b70d644e63	key
1	00	1	19711975d62eb677a38fc1111a729c3a	plaintext
1	00	1	a367fd2cd197119b67738fc19f9b05d3	plaintext
1	00	1	dec67fd2cdb574cacd68e49f9b05d336	plaintext
1	00	1	36bdec6e44a50e16f59a44a574cacd68	plaintext

A.3. RSA512 Core

Given 512-bit plaintext X , 512-bit key Y and 512-bit modulus M , the RSA512 core [71] calculates the cipher text $s = x^y \text{ mod } m$. The core receives the inputs serially. It has four 16-bit inputs: x , y , m and r_c . the r_c is the squared Montgomery constant modulo m . The Montgomery constant is $r = 2^{16 \cdot (32+1)}$ and $r_c = r^2 \text{ mod } m$. The core receives the four 512-bit input serially with a 16-bit word at a time. It outputs the 512-bit output serially with a 16-bit word at a time.

The RSA512 core has additional input signal $start_in$. It should be set active when loading the first 16 bits of m . After 6 cycles $valid_in$ is used to feed the 512-bit of the rest of the data serially. The signal usage is somehow complicated as seen in the Verilog test bench. Although it is possible to build a wrapper for this core, we decided to remove this signal to make the core interface better. We build an FSM to generate the $start_in$ signal on the starting of the computation. The new top-level module of the RSA512 Verilog code is as follow:

```

module rsa_top2(input clk, reset, valid_in,input [15:00] x, y, m, r_c,
bit_size,output [15:00] s,output valid_out);
    reg [2:0] state = 3'h0;
    reg start_in = 1'b0;
    reg rst = 1'b0;
    reg [3:0] cnt = 4'h0;
    always @(posedge clk, posedge reset)
        if (reset)
            state = 3'h0;
        else case (state)
            3'h0: begin start_in <= 1'b0; rst <= 1'b1; cnt <= 4'h2; state =
3'h1; end
            3'h1: begin cnt <= cnt - 1'b1; if (cnt == 1'b0) state = 3'h2; end
            3'h2: begin rst <= 1'b0; cnt <= 4'ha; state = 3'h3; end
            3'h3: begin cnt <= cnt - 1'b1; if (cnt == 1'b0) state = 3'h4; end
            3'h4: begin start_in <= 1'b1; state = 3'h5; end
            3'h5: begin start_in <= 1'b0; state = 3'h6; cnt <= 4'h8; end
            3'h6: begin cnt <= cnt - 1'b1; if (cnt == 1'b0) state = 3'h7; end
            3'h7: state = 3'h7;
            default: state = 3'h0;
        endcase
    rsa_top (.clk(clk), .reset(rst), .valid_in(valid_in),
.start_in(start_in), .x(x), .y(y), .m(m), .r_c(r_c), .s(s),
.valid_out(valid_out), .bit_size(bit_size));
endmodule

```

The actual inputs and outputs of the core are depicted in Figure 8.3. The figure also shows an XML description for a defined input and output groups for that core. There two input groups. The first group contains the reset signal and the bit_size constant. The second group contains the plaintext x , the key y , the modulus m , the squared Montgomery constant r_c and the valid signal. There is one output group contains the cypher text and the valid signal.

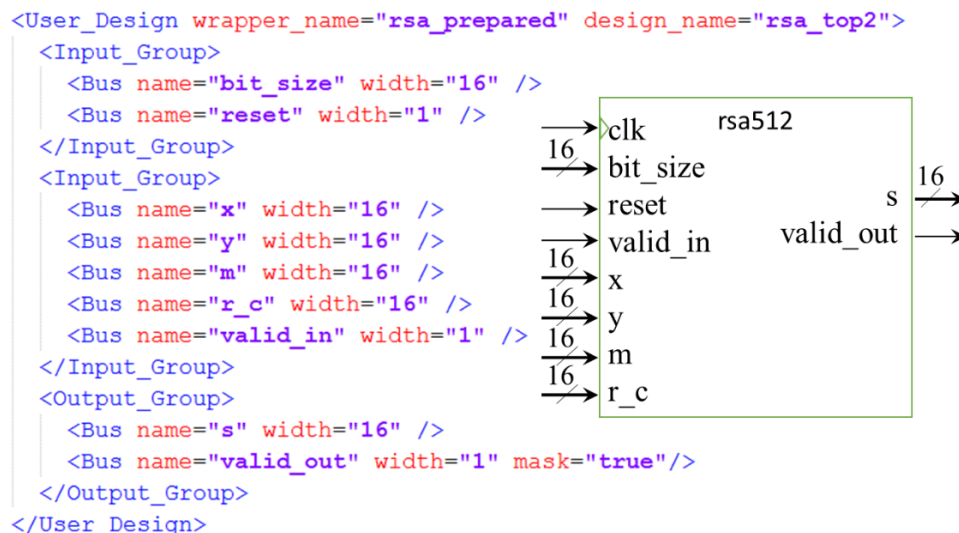


Figure 8.3: The XML specification of the input/output groups of the rsa512 core [71].

Table 8.3: Snapshot of the serializer’s output for the RSA512. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register. It then set the value of the modulus m . Then, it sets the *bit_size* constants. After that, it sets the clock counter to 32 cycles. Then, it applies those clocks to the design. After that, it resets the clock counter to zero. Then, the data starts.

CLK	SEL	65-bit data					Description
			r_c	m	y	x	
0	11	0	ffff	ffff	ffff	ffff	mask
0	01	0	f579	b491	42b1	f3ad	set m
1	00	0	0000	0000	0001	0200	set bit_size
0	10	0	0000	0000	0000	0020	clock counter
1	00	0	0000	0000	0000	0200	apply clocking
0	10	0	0000	0000	0000	0000	clock counter
1	01	1	f579	b491	42b1	f3ad	data
1	01	1	6ee9	1417	1ad3	8e40	data
1	01	1	972d	b498	a827	6af9	data

A.4. FDCT Core

The DCT core [69] transforms the image data to a different domain using the cosine function. It decomposes the signal into underlying spatial frequencies. The DCT transform is invertible. It is used in image compression since neighboring pixels within an image tend to be highly correlated. The JPEG Encoder core explained above contains a DCT core.

The core has 8-bit input and produces 12-bit output. It has a enable signal *ena* that should be high when computation starts. The core has a data strobe input *dstrb* that indicates the starting of the input data. It has a data strobe output *den* that indicates the starting of the output. Since these strobes are pulses we decide to convert them to valid signals that strobes all the inputs and outputs. The new top-level module of the RSA512 Verilog code is as follow:


```

module fdct_prepared #(parameter di_width = 8, do_width = 12)
  (input clk, input ena, input rst, input dstrb, input [di_width-1:0]
  din, output [do_width-1:0] dout, output douten);
  wire douten1;
  fdct #(.di_width(di_width), .do_width(do_width))
  fdct_inst (clk, ena, rst, dstrb, din, dout, douten1);
  reg [7:0] cntr = 7'b0;
  assign douten = (cntr != 7'b0);
  always @(posedge clk, negedge rst)
    if (!rst)
      cntr <= 7'b0;
    else if (douten1)
      cntr <= 7'd64;
    else if (douten)
      cntr <= cntr - 1'b1;
endmodule

```

The actual inputs and outputs of the core are depicted in Figure 8.4. The figure also shows an XML description for a defined input and output groups for that core. The core has two input groups. The first group has the least frequently used signals. The second group has the 8-bit data input. The core has one output group contains the 8-bit output and the valid signal.

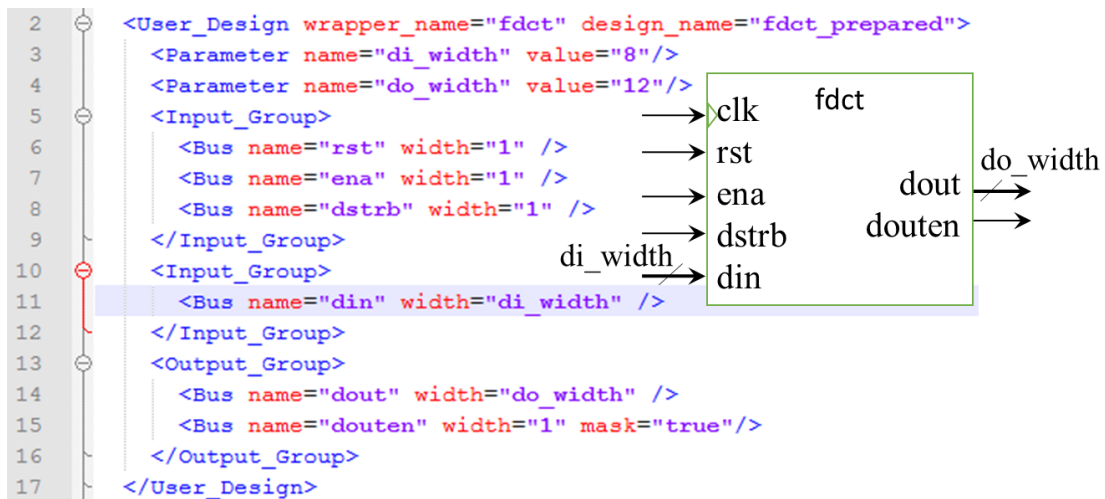


Figure 8.4: The XML specification of the input/output groups of the DCT core [69].

Table 8.4: Snapshot of the serializer’s output for the FDCT. Each row in the table represents one input group. The CLK column indicates whether to apply a clocking at that input or not. The SEL column represents the input group index. The table starts by setting values to the mask register. After that, it resets the clock counter to zero. Then, it sets the enable and reset signals high and the dstrb signal low. After that, the first byte of the data is set. Then, it sets the dstrb signal high. After that, it sets the dstrb signal low again. Then, the data continues

CLK	SEL	8-bit data	Description
0	11	255	mask
0	10	0	clock counter
1	00	0	dstrb=0,ena=1,rst=1
0	01	11	data
1	00	0	dstrb=1,ena=1,rst=1
1	00	0	dstrb=0,ena=1,rst=1
1	01	16	data
1	01	21	data
1	01	25	data

A.5. JPEG Images Edge Detection

This hardware was made by combining several cores [72, 70, 78, 77, 76]. It receives encrypted JPEG image, decrypts it using the AES-CTR, decompresses it to a BMP image using the decompress core [77], detects its edges using the Canny edge detection core [78], encodes the resulted image to a JPEG image using the JPEG encoder [70] and decrypts it using the AES-CTR. The core has 64-bit input *data_in*, reset signal *reset* and start signals *start*. The start signal is used to start the AES-CTR counter since it needs 20-cycle initialization then it becomes able to produce one output per clock. It has 64-bit output *data_out* with a valid signal *valid_out*.

The inputs and outputs of the design are depicted in Figure 8.5. The figure also shows an XML description for a defined input and output groups for that core. The core has two input groups. The first group has the least frequently used signals *start* and *Reset*. The second group has the 64-bit data input. The core has one output group contains the 64-bit output and the valid signal.

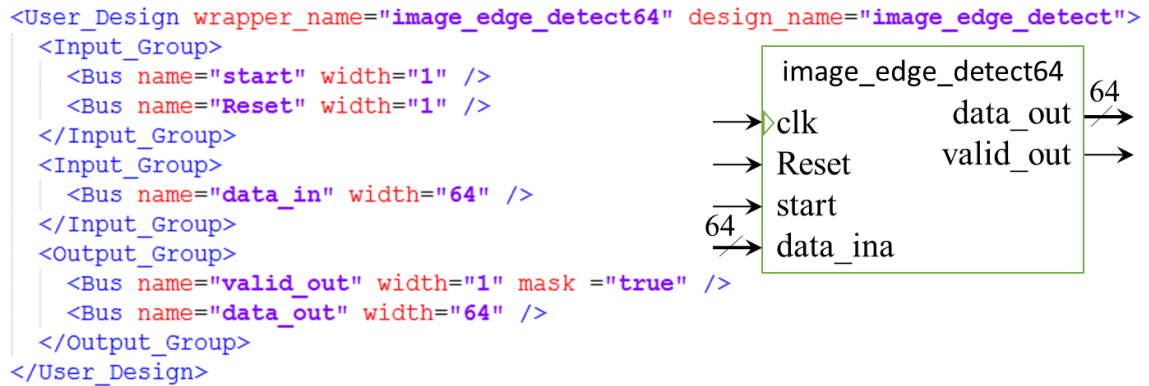


Figure 8.5: The XML specification of the input/output groups of the image edge detection we designed by combining several cores.

A.6. Decrypt-Compute-Encrypt

To have secure computation, we assume that a decrypter and an encrypter should be integrated within the application hardware itself. We use two instances of the AES-CTR core for decryption and encryption. The hardware input is the first AES-CTR input. The hardware output is the second AES-CTR output.

The inputs and outputs of the design are depicted in Figure 8.6. The figure also shows an XML description for a defined input and output groups for that core. The core has three input groups for the 128-bit plaintext *state*, the 128-bit key *key*, and the reset signal *reset*. The core has one output group contains the 128-bit output *out*.

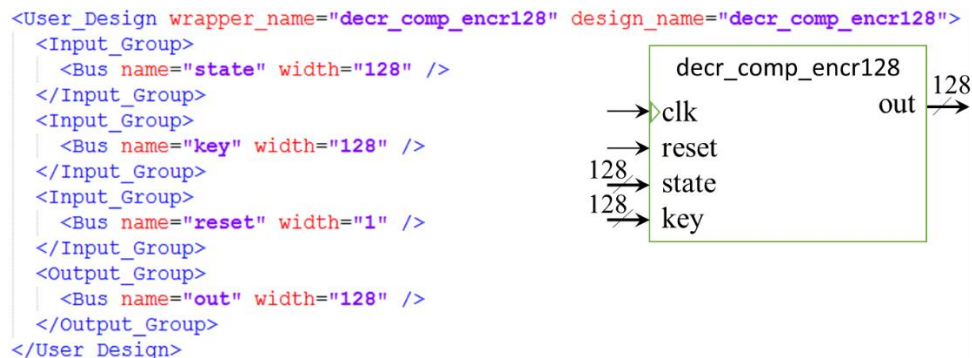


Figure 8.6: The XML specification of the input/output groups of the decrypt-compute-decrypt hardware designed using the AES-CTR which uses the AES-ECB core [72].

B. Software tool for Ethernet packet generation and platform test bench

For simulation purposes, we have designed a software tool that for a given user data it generates the corresponding Ethernet packets. The tool also generates the Verilog test bench for the whole platform that feeds the generated Ethernet packets to the core to simulate its exact running. In this section, we list and explain the algorithms used in this software tool.

Algorithm 3 Generate platform test bench

```
1: procedure GENERATEPLATFORMTESTBENCH(data_file, benchmark_name, data_bus_width,  
   vFPGA_index)  
2:   benchmark_file = data_file  
3:   VerilogCode = VerilogCode + test_bench_header(data_bus_width)  
4:   VerilogCode = VerilogCode + Instantiate_the_Platform_top_module(data_bus_width)  
5:   VerilogCode = VerilogCode + fpgaclk_and_RXCLK_registers(data_bus_width)  
6:   VerilogCode = VerilogCode + write_output_to_file_register()  
7:   VerilogCode = VerilogCode + initialBlock_header()  
8:   data_per_packet = 1408  
9:   no_of_packets = FileLength(benchmark_file)/data_per_packet  
10:  for i = 1 To no_of_packets do  
11:    user_data = GetFromFile(benchmark_file, data_per_packet)  
12:    packet_data = makeUdpHeader(vFPGA_index, data_per_packet, 6)  
13:    packet_data = packet_data + myHeader(i)  
14:    packet_data = packet_data + user_data  
15:    packet_data = packet_data + CalcCRC(packet_data)  
16:    L = length(packet_data)  
17:    VerilogCode = VerilogCode + StartPacketCode()  
18:    for j = 1 To L do  
19:      VerilogCode = VerilogCode + PacketWordCode(j)  
20:    end for  
21:    VerilogCode = VerilogCode + EndPacketCode()  
22:    VerilogCode = VerilogCode + Add_interpacket_wait_statement()  
23:  end for  
24:  VerilogCode = initialBlock_footer()  
25:  VerilogCode = test_bench_footer()  
26:  Return(VerilogCode)  
27: end procedure
```

Figure 8.7: Generate platform test bench.

The algorithms used for the packet and test bench generation are listed in Figure 8.7, Figure 8.8, Figure 8.9 and Figure 8.10. It is a template-based Verilog code generator. It starts by adding the Verilog code header in line 3 and an instantiation of the platform top module in line 4. In Line5, it adds the clocking registers that generate the FPGA clock

(usually 100MHz) and the Ethernet receiver clock. In line 6, it adds the output register which writes the design outputs to a file. In line 7, the initial block in the testbench starts. In line 8, the packet payload size is determined as 1408 bytes.

The loop in line 10 generates a UDP packet for every 1408 bytes of the data file. The loop starts by reading the data portion from the file, generating a UDP header for it, adding an additional header that includes statistics counters and finally calculating the cyclic redundancy check (CRC) of the Ethernet packet. After generating each packet, it is converted to Verilog test bench lines added to the initial block which we have already started in line 7. The loop in line 18 adds a Verilog line for each data word of the Ethernet packet. In line 22, a Verilog wait statement is added that force the simulator not to start receiving a new packet if its receiving buffer is full (i.e. the user hardware is busy doing computation on previous data). The rest of the lines adds the footers of the initial block which we have already started in line 7 and the test bench module which we have already started in line 3.

Algorithm 4 Make UDP Header

```

1: procedure MAKEUDPHEADER(vFPGA_index, data_per_packet, myHeaderSize)
2:   udp_length = data_per_packet + myHeaderSize + 8
3:   total_len = 20 + udp_length
4:   id = 21468//generatearbitraryid
5:   EthHeader: packet = DEST_MAC(vFPGA_index) + SRCE_MAC(vFPGA_index) +
   "0800"
6:   IPHeader:   packet   = packet + "4500" + total_len + "53DC00008011" +
   checksum(id, total_len, DEST_IP(vFPGA_index), SRCE_IP(vFPGA_index))   +
   SRCE_IP(vFPGA_index) + DEST_IP(vFPGA_index)
7:   UDPHeader:  packet   = packet + SRCE_PORT(vFPGA_index) +
   DEST_PORT(vFPGA_index) + udp_length + checksum_udp(udp_length,
   SRCE_IP(vFPGA_index), DEST_IP(vFPGA_index), SRCE_PORT(vFPGA_index),
   DEST_PORT(vFPGA_index))
8:   Return(packet)
9: end procedure

```

Figure 8.8: Make UDP header algorithm. The algorithm is inspired by the IP formal definition in [RFC 791](#) [83] and the UDP formal definition in [RFC 768](#) [84].

Algorithm 6 Calculate udp checksum

```
1: procedure CHECKSUMUDP(udp_length, DEST_IP, SRCE_IP, SRCE_PORT,  
   DEST_PORT)  
2:    $m = 17 + \text{rightPart}(\text{DEST\_IP}) + \text{leftPart}(\text{DEST\_IP}) + \text{rightPart}(\text{SRCE\_IP}) +$   
    $\text{leftPart}(\text{SRCE\_IP}) + \text{SRCE\_PORT} + \text{DEST\_PORT} + \text{udp\_length}$   
3:   while  $\text{size}(m) > 4\text{bytes}$  do  
4:      $m = \text{rightPart}(m) + \text{leftPart}(m)$   
5:   end while  
6:   Return(m)  
7: end procedure
```

Figure 8.9: Calculate IP checksum algorithm. The algorithm is inspired by the IP checksum calculation description.

Algorithm 6 Calculate udp checksum

```
1: procedure CHECKSUMUDP(udp_length, DEST_IP, SRCE_IP, SRCE_PORT,  
   DEST_PORT)  
2:    $m = 17 + \text{rightPart}(\text{DEST\_IP}) + \text{leftPart}(\text{DEST\_IP}) + \text{rightPart}(\text{SRCE\_IP}) +$   
    $\text{leftPart}(\text{SRCE\_IP}) + \text{SRCE\_PORT} + \text{DEST\_PORT} + \text{udp\_length}$   
3:   while  $\text{size}(m) > 4\text{bytes}$  do  
4:      $m = \text{rightPart}(m) + \text{leftPart}(m)$   
5:   end while  
6:   Return(m)  
7: end procedure
```

Figure 8.10: Calculate UDP checksum algorithm. The algorithm is inspired by the UDP checksum calculation description in the UDP formal definition in [RFC 768](#) [84].

C. A description of the implemented Verilog code

Figure 8.11 shows a snapshot of the Xilinx ISE design tab. It shows the hierarchy and the components of the virtualization platform. The hierarchy starts by clocking resources in the first four nodes of the top module. The virtualization module consists of two parts; the platform and the network controller. The platform contains data routers (mux and demux), the reconfiguration module (uses ICAP) and one vFPGA. Any number of vFPGA can be used by setting the No_of_USERS parameter in the top module. The whole platform is parametrizable.

The network controller contains an address table, session store, receiver, and transmitter. The address table contains the MAC and IP addresses of the vFPGAs and the static logic. It generates the vFPGA index for the received packet. The session store stores the client address extracted from the received packet. The receiver contains a packet sniffer and double buffer. The packet sniffer sets flags according to the received packet such as; is it a UDP, TCP or ARP packet. The double buffer stores the payload of the received packet if its address gets a match with the address table. The double buffer produces the received payload while it receives another payload. It must wait until receiving the full packet with correct cyclic redundancy check (CRC). The network transmitter contains the packetizing module which contains a finite state machine that produces the Ethernet header, followed by IP header if needed, followed by the selected header. Currently, three types of network packets can be produced; ARP replay, ICMP replay, and UDP packet.

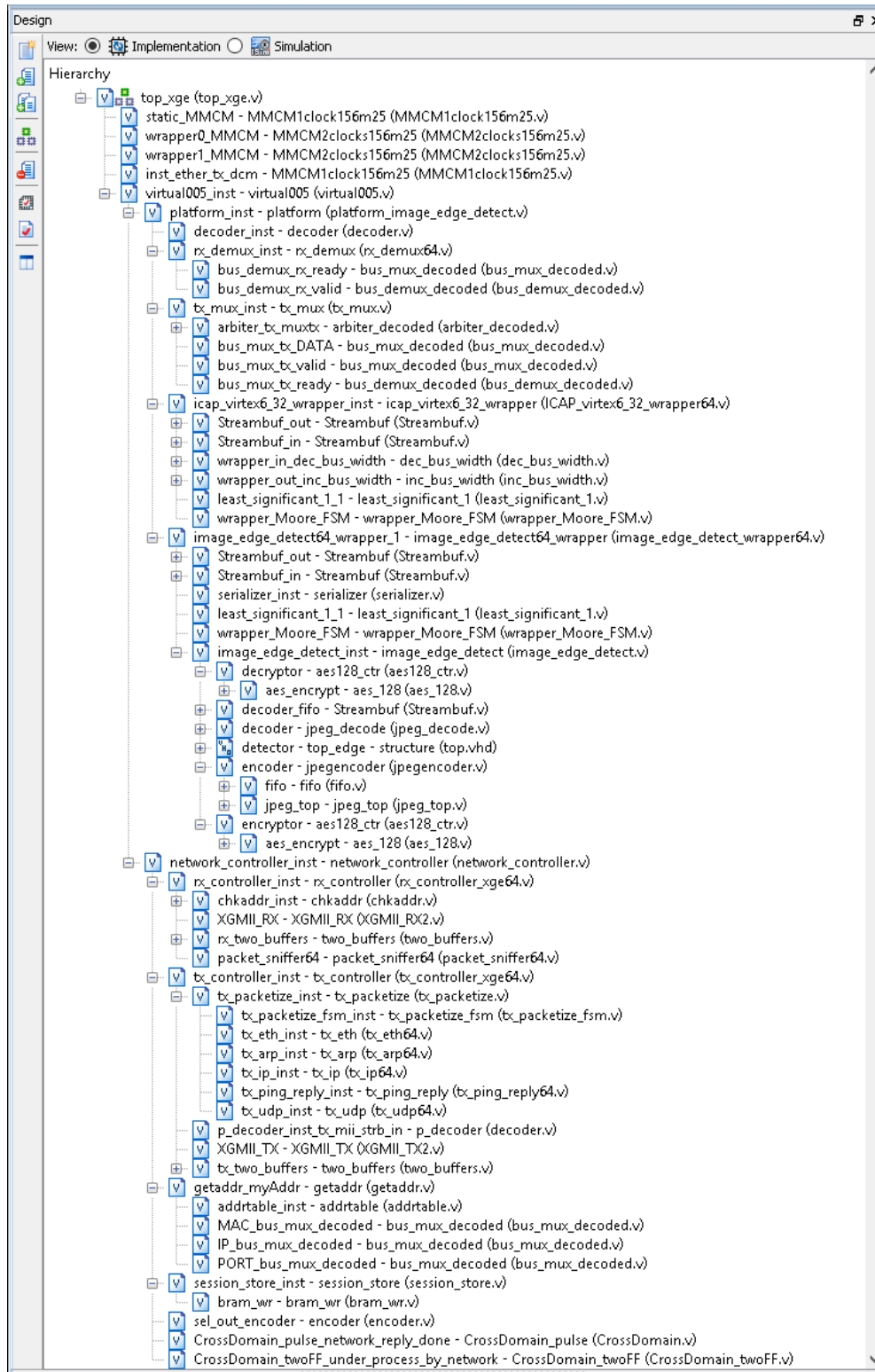


Figure 8.11: A snapshot shows the hierarchy and components of an implemented version of the virtualization platform. The hierarchy starts the root node “top_xge” which contains clocking resources appears in the first four nodes and the virtualization module “virt005” which contains the platform and the network controller. The platform contains data routers (mux and demux), the reconfiguration module (uses ICAP) and one vFPGA. The vFPGA contains the image_edge_detect application which uses four already-made cores; AES_128 [72], jpeg_decode [77], top_edge [78] and jpeg encoder [70].

References

- [1] "OpenPOWER Cloud," IBM, [Online]. Available: research.ibm.com/labs/china/supervessel.html. [Accessed 2018].
- [2] P. Gupta, "Bringing FPGA Acceleration to the Cloud," Intel, 20 March 2017. [Online]. Available: <https://itpeernetwork.intel.com/fpga-acceleration-to-the-cloud/>. [Accessed 2018].
- [3] M. Russinovich, "FPGAs and the New Era of Cloud-based 'Hardware Microservices'," Microsoft, 8 June 2017. [Online]. Available: <https://thenewstack.io/developers-fpgas-cloud/>. [Accessed 2018].
- [4] C. Brugger, L. Dal'Aqua, J. A. Varela and C. De Schryver, "A quantitative cross-architecture study of morphological image processing on CPUs, GPUs, and FPGAs," in *Computer Applications & Industrial Electronics (ISCAIE), 2015 IEEE Symposium on*, Langkawi, Kedah, Malaysia, 2015.
- [5] BERTEN, "Gpu vs fpga performance comparison," [Online]. Available: http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf. [Accessed 2018].
- [6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, pp. 10-22, 2015.
- [7] Amazon, "Amazon EC2 F1 Instances," [Online]. Available: aws.amazon.com/ec2/instance-types/f1/. [Accessed 2018].
- [8] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Gaithersburg, 2011.
- [9] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*, 2014.

- [10] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang and K. Wang, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers - CF '14*, 2014.
- [11] N. Tarafdar, N. Eskandari, T. Lin and P. Chow, "Designing for FPGAs in the Cloud," *IEEE Design & Test*, pp. 1-1, 2017.
- [12] J. Weerasinghe, F. Abel, C. Hagleitner and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *Proceedings - 2015 IEEE 12th International Conference on Ubiquitous Intelligence and Computing, 2015 IEEE 12th International Conference on Advanced and Trusted Computing, 2015 IEEE 15th International Conference on Scalable Computing and Communications*, 20, 2016.
- [13] "OpenStack service overview," OpenStack, 19 January 2019. [Online]. Available: https://docs.openstack.org/security-guide/_images/marketecture-diagram.png. [Accessed 2019].
- [14] "Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide," Intel, 2018. [Online]. Available: <https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html>.
- [15] Xilinx, "Virtex Series Configuration Architecture Series Configuration Architecture," 20 October 2004. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp151.pdf. [Accessed 2019].
- [16] Xilinx, "Virtex-6 FPGA Configuration User Guide," 18 November 2015. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug360.pdf. [Accessed 2019].
- [17] Xilinx, "Vivado Design Suite User Guide Partial Reconfiguration," 27 April 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf. [Accessed 2019].
- [18] T. Lu, R. Kenny and S. Atsatt, "Secure Device Manager for Intel Stratix 10 Devices Provides FPGA and SoC Security," [Online]. Available: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01252-secure-device-manager-for-fpga-soc-security.pdf>. [Accessed 2019].

- [19] Altera, "Partial Reconfiguration IP Core," 4 5 2015. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_partrecon.pdf. [Accessed 2019].
- [20] Intel, "Creating a Partial Reconfiguration Design," 4 January 2019. [Online]. [Accessed 2019].
- [21] Intel, "Design Planning for Partial Reconfiguration," 4 November 2013. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/qts_qii51026.pdf. [Accessed 2019].
- [22] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1-4, 2013.
- [23] "The open standard for parallel programming of heterogeneous systems," Khronos, [Online]. Available: <https://www.khronos.org/opencl/>. [Accessed 2018].
- [24] "SDAccel Development Environment," Xilinx, [Online]. Available: <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>. [Accessed 2018].
- [25] "Vivado High-Level Synthesis," Xilinx, [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>. [Accessed 2018].
- [26] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," *FPGA '11 Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33-36, 27 2 2011.
- [27] J. Choi, S. Brown and J. Anderson, "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs," in *International Conference on Field-Programmable Technology (FPT)*, Kyoto, Japan , 2013.
- [28] D. Koch, F. Hannig and D. Ziener, "FPGA Versus Software Programming: Why, When, and How?," in *FPGAs for Software Programmers*, Switzerland, Springer, 2016, pp. 1-21.

- [29] H. K.-H. So and C. Liu, "FPGA Overlays," in *FPGAs for Software Programmers*, New York, Springer, 2016, pp. 285-306.
- [30] T. Bollengier, M. Najem, J.-C. L. Lann and L. Lagadec, "Demo: Overlay architectures for heterogeneous FPGA cluster management," *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 239-240, 2016.
- [31] A. Brant and G. G. F. Lemieux, "ZUMA: An open FPGA overlay architecture," *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 93-96, 4 2012.
- [32] D. Koch, C. Beckhoff and G. G. F. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1-8, 9 2013.
- [33] D. Capalija and T. Abdelrahman, "A coarse-grain fpga overlay for executing data flow graphs," in *The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2012)*, 2012.
- [34] X. Li, A. K. Jain, D. L. Maskell and S. A. Fahmy, "An Area-Efficient FPGA Overlay using DSP Block based Time-multiplexed Functional Units," in *2nd International Workshop on Overlay Architectures for FPGAs (OLAF2016)*, Monterey, CA, USA, 2016.
- [35] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant FPGA compilation," *IEEE Embedded Systems Letters*, vol. 3, no. 3, pp. 81-84, 12 9 2011.
- [36] S. A. Fahmy, K. Vipin and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015*, 2016.
- [37] O. Knodel and R. G. Spallek, "Computing framework for dynamic integration of reconfigurable resources in a cloud," *2015 Euromicro Conference on Digital System Design (DSD)*, pp. 337-344, 1 8 2015.
- [38] J. Weerasinghe, F. Abel, C. Hagleitner and A. Herkersdorf, "Disaggregated fpgas: Network performance comparison against bare-metal servers, virtual machines and linux containers," *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pp. 9-17, 12 12 2016.

- [39] J. Weerasinghe, R. Polig, F. Abel and C. Hagleitner, "Network-Attached FPGAs for Data Center Applications," *IEEE International Conference on Field-Programmable Technology (FPT '16)*, 2 2016.
- [40] H. L. Kidane, E.-B. Bourennane and G. Ochoa-Ruiz, "NoC Based Virtualized Accelerators for Cloud Computing," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Lyon, France, 2016.
- [41] A. Putnam and Others, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *ISCA'14*, 2014.
- [42] Q. Zhao, M. Amagasaki, M. Iida, M. Kuga and T. Sueyoshi, "Enabling FPGA-as-a-Service in the Cloud with hCODE Platform," *IEICE Transactions on Information and Systems*, vol. 101, no. 2, pp. 335--343, 2018.
- [43] M. Jacobsen, D. Richmond, M. Hogains and R. Kastner, "RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, pp. 22:1--22:23, 2015.
- [44] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1-7, 9 2014.
- [45] C. Vatsolakis and D. Pnevmatikatos, "RACOS: Transparent access and virtualization of reconfigurable hardware accelerators," *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 11-19, July 2017.
- [46] O. Knodel, P. R. Genssler and R. G. Spallek, "Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures," *CENICS 2017 : The Tenth International Conference on Advances in Circuits, Electronics and Microelectronics*, vol. 2, pp. 33-38, 2017.
- [47] O. Knodel, P. R. Genssler and R. G. Spallek, "Migration of long-running Tasks between Reconfigurable Resources using Virtualization," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 56-61, 2017.
- [48] ARM, "AMBA AXI4-Stream Protocol Specification," [Online]. Available: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ih0051a/index.html>. [Accessed 2018].

- [49] E. Izenberg, N. Bshara, C. Pettey and C. K. OHRT, "Fpga-enabled compute instances". Washington Patent WO2017117122A1, 6 7 2017.
- [50] M. Asiatici, N. George, K. Vipin, S. A. Fahmy and P. Ienne, "Designing a virtual runtime for FPGA accelerators in the cloud," *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1-2, 8 2016.
- [51] M. Asiatici, N. George, K. Vipin, S. A. Fahmy and P. Ienne, "Virtualized Execution Runtime for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, pp. 1900-1910, 2017.
- [52] "The OpenCL Specification Version 2.2-8," Khronos OpenCL Working Group, 8 October 2018. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf. [Accessed 2018].
- [53] Xilinx, "SDAccel Development Environment," Xilinx, [Online]. Available: www.xilinx.com/products/design-tools/software-zone/sdaccel.html. [Accessed 2018].
- [54] "Intel SDK for OpenCL Applications," Intel, 2018. [Online]. Available: <https://software.intel.com/en-us/intel-opencl>.
- [55] "AWS Shell Interface Specification," Amazon, [Online]. Available: https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md. [Accessed 2018].
- [56] "Amazon FPGA Image (AFI) Management Tools," Amazon, [Online]. Available: https://github.com/aws/aws-fpga/blob/master/sdk/userspace/fpga_mgmt_tools/README.md. [Accessed 2018].
- [57] IBM, "SuperVessel, an OpenPOWER cloud platform," IBM Research – China, [Online]. Available: <http://research.ibm.com/labs/china/supervessel.html>. [Accessed 2018].
- [58] W. Wang, M. Bolic and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1-9, 2013.
- [59] O. Sander, S. Baehr, E. Luebbers, T. Sandmann, V. V. Duy and J. Becker, "A flexible interface architecture for reconfigurable coprocessors in embedded multicore

- systems using PCIe Single-root I/O virtualization," in *International Conference on Field-Programmable Technology (FPT)*, Shanghai, China, 2014.
- [60] D. V. Vu, O. Sander, T. Sandmann, S. Baehr, J. Heidelberger and J. Becker, "Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using PCI express single-root I/O virtualization," in *International Conference on ReConfigurable Computing and FPGAs (ReConFig14)*, Cancun, Mexico, 2014.
- [61] D. Theodoropoulos, N. Alachiotis and D. Pnevmatikatos, "Multi-FPGA Evaluation Platform for Disaggregated Computing," *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 193-193, April 2017.
- [62] I. Magaki, M. Khazraee, L. Vega and M. B. Taylor, "ASIC clouds: specializing the datacenter," *ISCA '16 Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 178-190, 2016.
- [63] S. A. Byma, "Virtualizing FPGAs for Cloud Computing Applications," University of Toronto, 2014.
- [64] M. Vesper, D. Koch, K. Vipin and S. A. Fahmy, "JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication," *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1-9, 2016.
- [65] "TCP/UDP/IP Network Protocol Accelerator," MLE, [Online]. Available: <https://www.missinglinkelectronics.com/index.php/menu-products/menu-network-protocol-accelerator>. [Accessed 2018].
- [66] F. Haque, J. Michelson and J. Michelson, *The Art of Verification with VERA, Verification Central*; 1 edition (September 1, 2001), 2001.
- [67] "Clock Control Block (ALTCLKCTRL) IP Core User Guide," Intel Altera, 4 4 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altclock.pdf. [Accessed 2018].
- [68] "UltraScale Architecture Clocking Resources User Guide," Xilinx, 19 December 2018. [Online]. Available:

- https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf. [Accessed 2018].
- [69] R. Herveille and A. Henson, "Video compression systems," 23 June 2009. [Online]. Available: https://opencores.org/projects/video_systems. [Accessed 2018].
- [70] D. Lundgren, "JPEG Encoder Verilog," 17 March 2010. [Online]. Available: <https://opencores.org/projects/jpegencode>. [Accessed 2018].
- [71] J. Castillo Villar, "An open-source implementaion of the 512 bit RSA algorithm," 12 January 2011. [Online]. Available: https://opencores.org/projects/rsa_512. [Accessed 2018].
- [72] H. Hsing, "tiny_aes," 14 December 2015. [Online]. Available: https://opencores.org/project/tiny_aes. [Accessed 2018].
- [73] R. Kirchgessner, G. Stitt, A. George and H. Lam, "VirtualRC: a virtual FPGA platform for applications and tools portability," *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 205-208, 22-24 February 2012.
- [74] G. McMillan, "Socket Programming," Python Software Foundation, 8 November 2018. [Online]. Available: <https://docs.python.org/2/howto/sockets.html>. [Accessed 2018].
- [75] D. Litzemberger, "PyCrypto - The Python Cryptography Toolkit," [Online]. Available: <https://www.dlitz.net/software/pycrypto/>. [Accessed 2018].
- [76] H. Ishihara, "JPEG Decoder," 13 March 2018. [Online]. Available: <https://opencores.org/projects/djpeg>, <http://www.pudn.com/Download/item/id/521458.html>. [Accessed 2018].
- [77] H. Ishihara, "JPEG Decoder," 24 April 2015. [Online]. Available: https://github.com/aquaxis/IPCORE/tree/master/aq_axi_djpeg. [Accessed 2018].
- [78] "Canny Edge Detector," 9 January 2014. [Online]. Available: https://opencores.org/projects/canny_edge_detector. [Accessed 2018].
- [79] "OpenCV (Open Source Computer Vision Library)," OpenCV team, [Online]. Available: <https://opencv.org/>. [Accessed 2018].

- [80] D. Jayasinghe, R. Ragel, J. A. Ambrose, A. Ignjatovic and S. Parameswaran, "Advanced modes in AES: Are they safe from power analysis based side channel attacks?," *IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 173-180, 19-22 October 2014.
- [81] M. Leonhard, "Estimate the latency from your browser to each AWS region," Amazon, 2010. [Online]. Available: <https://www.cloudping.info/>. [Accessed 2018].
- [82] "Partial Reconfiguration User Guide," Xilinx, 3 May 2010. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf. [Accessed 2018].
- [83] V. Cerf and B. Kahn, "INTERNET PROTOCOL," September 1981. [Online]. Available: <https://tools.ietf.org/html/rfc791>. [Accessed 2018].
- [84] D. P. Reed, "RFC 768 : User Datagram Protocol," 28 August 1980. [Online]. Available: <https://tools.ietf.org/html/rfc768>. [Accessed 2018].
- [85] Altera, "Partial Reconfiguration IP Core," 8 May 2017. [Online]. Available: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-partrecon.pdf>. [Accessed 2019].

Vitae

Name : Amran Abdulrahman Al-aghbari

Nationality : Yemen

Date of Birth : 29-12-1978

Email : emran.hsb@hotmail.com, emran.hsb@gmail.com



Academic Background : Amran received his BS.C in Computer Science, from Sana'a University, Sana'a, Yemen, in June 2004. He worked as a lecturer in computer science department, Taiz University, Taiz, Yemen from 2005 to 2009. He received his M.Sc. degree in Computer Engineering, from KFUPM, Saudi Arabia, in December 2012. He defended his PhD in Computer Science and Engineering from KFUPM, Saudi Arabia, in December 2018. Amran interest includes programming and designing systems and tools, hardware/software co-design, hardware design languages, HLL-to-HDL compilers, HDL editing tools, virtualized reconfigurable computing.

Publications from the dissertation:

- [1] A. Al-Aghbari and M. E. S. Elrabaa, "A platform for FPGA virtualization in clouds and data centers," *Microprocessors and Microsystems*, vol. 62, pp. 61-71, 2018.
- [2] A. Al-Aghbari and M. E. S. Elrabaa, " Cloud-Based Secure FPGA Custom Computing Machines for Streaming Applications," UNDER PREPARATION